TECHNISCHE UNIVERSITÄT CHEMNITZ

Fakultät für Informatik

Professur Technische Informatik

Masterarbeit

Konzeption und Implementierung eines Werkzeugs für den Test von AUTOSAR Applikationen mit Intra-ECU Kommunikation

eingereicht von:
Markus Leib

Chemnitz, den 30. März 2016

Prüfer: Prof. Dr. Wolfram Hardt
Betreuer: Dipl.-Inf. Norbert Englisch

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die Möglichkeit, komplexe Applikationen für Steuergeräte zu entwerfen und zu implementieren wird durch die Systemarchitektur AUTOSAR stark vereinfacht. Ein Applikations-Entwickler implementiert nur noch die reine Funktionalität, während der Teil, der die Interaktion mit der Hardware ermöglicht, von entsprechenden Werkzeugen automatisch generiert wird. Viele Hersteller bieten hierfür Produkte, die vom grafischen Entwurf der Software bis zu finalen Tests des fertigen Steuergeräte-Codes alles abdecken. Zentrales Element ist hierbei immer die AUTOSAR-XML, welche alle Informationen zur Anwendung aufnimmt. Dadurch können verschiedene Zulieferer einzelne Module zu einer Anwendung beisteuern. Durch den technologischen Fortschritt und die Integrierbarkeit von verfügbaren Teilsystemen oder einzelnen Komponenten werden eingebettete Systeme immer komplexer ([Har02]). Daher müssen neue Spezifikationen erstellt und demzufolge neue AUTOSAR-Versionen publiziert werden. Die Hersteller müssen sich an diese neuen Vorgaben halten, um so konkurrenzfähig auf dem Markt zu bleiben ([ZS14]).

1.1 Motivation

Im Entwicklungsprozess einer AUTOSAR-Applikation spielt das Testen der Anwendung eine entscheidende Rolle. Um Zeit und Geld einzusparen wird der zu testende Quellcode hardwareunabhängig ausgeführt. Dies wird durch eine sogenannte virtuelle Absicherungs-Plattform (VAP) realisiert. Die Richtlinien des AUTOSAR-Schichtenmodells erlauben eine enorme Vereinfachung dieser Herangehensweise. Durch die einzelnen Layer im Modell kann der Testprozess noch früher gestartet werden, indem nur die Applikation überprüft wird. Namhafte Hersteller von Testsoftware verwenden dafür einen eigenes entwickelten Code-Generator, um die Kommunikation der Softwarekomponenten zu realisieren ([ZS14]).

An der Professur Technische Informatik der TU Chemnitz wird mit dem Forschungsdemonstrator für den „Automatisierten Test von AUTOSAR Systemen“ (ASTAS) das Ziel verfolgt, die Schichten im AUTOSAR-Modell einzeln zu prüfen. Das soll sowohl mit statischen Analyse-Verfahren als auch mit dynamischen Tests realisiert werden. Diese Arbeit befasst sich folglich mit dem statischen und dynamischen Prüfen der Applikation und soll dementsprechend in ASTAS integriert werden.

1.2 Struktur der Arbeit

Diese Arbeit untergliedert sich in die Kapitel Grundlagen, Stand der Technik, Konzept, Implementierung und Testergebnisse.

Zuerst wird dem Lesenden im Kapitel Grundlagen die Fähigkeit verliehen, den Inhalt der Arbeit nachvollziehen zu können. Dabei wird auf die Hauptmerkmale von AUTOSAR eingegangen und die wichtigsten Bestandteile einer Applikation anhand von Code-Listings erläutert. Anschließend folgt die Beschreibung der für diese Arbeit relevanten Basissoftwaremodule. Weil Testen ebenfalls eine wichtige Rolle in dieser Arbeit einnimmt, werden auch hierzu alle erforderlichen Basics vermittelt.

Im zweiten Kapitel werden die aktuellen Herangehensweisen bei dem Test von AUTOSAR-Steuergeräte-Software vorgestellt. Anschließend folgt eine Analyse der erforderlichen Programme für die Simulation von Steuergeräte-Software. Hierbei ist das Ziel, die Applikation ohne Basissoftware zu testen. Eine Erläuterung der hierfür erforderlichen Kriterien schließt das Kapitel ab.

Das Konzept dieser Arbeit wird durch das dritte Kapitel erläutert. Anhand von einzelnen Phasen, die zur Realisierung beitragen, wird dem Lesenden eine Leitlinie für das Nachvollziehen der Arbeit geboten. Anschließend folgt die Planung für ein mögliches Programm. Abgerundet wird das Kapitel mit sich ergebenden Einschränkungen bei diesem Konzept.

Für die Realisierung des Konzepts wird mit Kapitel fünf die Implementierung des Programms und die Trennung des Testobjektes auf Quellcode-Ebene präsentiert. Auch hierbei kann sich der Lesende an den einzelnen Phasen aus dem Konzept orientieren.

Im letzten Kapitel wird die entstandene Arbeit auf Herz und Nieren mit einer eigens erstellten Applikation geprüft.

2 Grundlagen

Dieses Kapitel bietet dem Leser eine fundamentale Wissensbasis um relevante Sachverhalte dieser Arbeit verstehen und nachvollziehen zu können. Vielmehr dient dieser Abschnitt auch als Nachschlagereferenz zum Verständnis bei entsprechenden Lösungsansätzen und Problemen. Das betrifft vor allem die Themen AUTOSAR und das generelle Testen von Software.

2.1 AUTOSAR

AUTOSAR ist eine offene und standardisierte Plattform für AUTOSAR Software Entwickler ([KF09]). Durch die Kapselung von Software zur Hardware, ist es möglich, automotive Anwendungen völlig Steuergeräte-Unabhängig zu entwickeln und zu testen.

> „Die Automotive Open System Architecture (AUTOSAR) Initiative, eine Konsortium, dem mittlerweile alle führenden Automobilhersteller und Zulieferer angehören, übernahm die Vorarbeiten von OSEK und HIS aus den 1990er Jahren und definiert heute eine vollständige Softwarearchitektur für Steuergeräte.“ [ZS14]

Durch diesen einheitlichen Standard, ist es möglich Software für Steuergeräte hardwareunabhängig zu entwickeln und zu Testen.

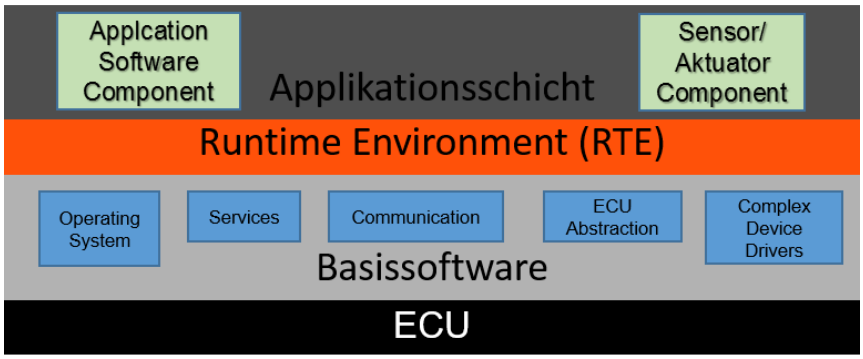

Abbildung 2.1: AUTOSAR MODEL nach Quelle: [AUT14h]

Das AUTOSAR Schichtenmodel bietet für den Einstieg einen groben Überblick und soll als Ausgangsbasis für weitere wichtige Aspekte dienen. Abbildung 2.1 zeigt den generellen Aufbau des AUTOSAR-Schichtenmodels. Die drei Schichten ermöglichen hierbei eine Abstraktion zur Hardware. Während die **Applikationsschicht** zur reinen Modellierung von Anwendungen betrachtet wird, ist die **Basissoftwareschicht** speziell für die Anpassung der Applikation an die Zielhardware nötig. Die Zwischenschicht, die so genannte **Run-Time-Environment** (RTE), stellt die erforderlichen Kommunikationswege zwischen Anwendungen untereinander und zur Basissoftware her ([KT13]).

2.1.1 AUTOSAR und Xml

XML ist eine der Technologien wenn es um einheitliches und standardisiertes Speichern oder Austauschen von Informationen geht. XML steht für Extensible Markup Language und kann alle möglichen Arten von Strukturierten Informationen Beschreiben. Z.B. Datenbanken oder Vectorgrafiken.

> „XML definiert eine Syntax, um strukturierte Datenbestände jeder Art mit einfachen, verständlichen Auszeichnungen zu verstehen, die zugleich von Anwendungen der unterschiedlichsten Art ausgewertet werden können.“ [Von09]

D.h. jeder Anwender kann aufgrund der Einfachheit von XML, eigene Dokumente erzeugen, verändern und auf jeder Plattform einsetzten.

Wohlgeformte XML Dokumente

XML Dokumente sind nach festen Regeln definiert, was es ermöglicht, ein standardisiertes abarbeiten der Informationen zu erzwingen. Damit eine XML-Datei „Wohlgeformt“ (gültig) ist, müssen 3 wesentliche Dinge erfüllt sein. Am Anfang der Datei steht immer die XML-Deklaration, es muss immer mindestens ein Datensatz vorhanden sein und die korrekte Verschachtlung der Tags, d.h. auch dass ein öffnender Tag immer auch mit einem Schließenden Tag beendet wird.[Von09].

Schemaprüfung

Die Bestimmung der Gültigkeit eines Dokuments, kann auch durch das Gegenprüfen eines Schemas erweitert werden. Dabei handelt es sich um eine Vorgabe, welche Informationen ein XML Dokument enthalten muss/darf. Diese Technik setzt eine zweite Datei voraus, in welcher die vorgeschriebenen **Tag**-Namen enthalten sind. Wurden hierfür früher noch DTD-Dateien (Dokumenttyp-Definition) verwendet, wird heute ein so genanntes Schema angewendet, da hierbei die Vorgaben auch in XML-Syntax realisiert werden.

AUTOSAR Xml

Bei der Entwicklung von AUTOSAR Applikationen gibt es Zahlreiche Programme und Testumgebungen, die für eine Stabile und Fehlerfreie Arbeit sorgen. Um die fertige Applikation auch in anderen Tools weiterverwenden zu können, z.B. für Basissoftwaregenerierung oder das Testen einer Anwendung, wird zur Beschreibung der Informationen XML verwendet. Nach AUTOSAR Spezifikation werden diese XML-Dokumente auch ARXML genannt. „AR“ ist das Akronym für AUTOSAR. Hierbei handelt es sich also um eine Standardisierte XML Datei. Durch die einheitliche Informationsbeschreibung kann die Datei bei allen Tool-Herstellern eingelesen werden.

```
<?xml version="1.0" encoding="utf-8"?>
<AUTOSAR xmlns="http://autosar.org/3.2.2" xsi:schemaLocation="http://autosar.org/3.2.2_autosar.xsd">
  <TOP-LEVEL-PACKAGES>
    <AR-PACKAGE>
        <SHORT-NAME/>
        ...
      </AR-PACKAGE>
      <AR-PACKAGE>
        <SHORT-NAME/>
        ...
      </AR-PACKAGE>
  </TOP-LEVEL-PACKAGES>
</AUTOSAR>
```

Listing 2.1: Ausschnitt aus einer ARXML-Datei

Das Listing zeigt die grundsätzliche Struktur des XML Dokuments. Zuerst wird die Datei mit der XML-Deklaration eingeleitet. Anschließend folgt der Beginn von AUTOSAR-spezifischen Elementen. Dieser Tag ist das Root-Elment und beinhaltet alle weiteren Informationen. Im Root-Tag wird der XML-Namespace (**xmlns**) und der Ort an dem die korrespondierende Schema-Datei (**xsi:schemalocation**) zu finden ist, festgelegt. Diese Angabe bezieht sich auf eine Internat-Adresse auf AUTOSAR.org und bietet dadurch die Möglichkeit, das Schema mit einem geeigneten Werkzeug automatisiert herunter zu laden und eine Validierung zu starten. Anhand des Namespace kann auch die Version von AUTOSAR erschlossen werden. Der nächste Knoten (**TOP-LEVEL-PACKAGES**) beschreibt die übergeordnete Struktur für alle folgenden Unter-Packages. Dabei können beliebige Packages in einem Projekt mit einem AUTOSAR-Modellierungs-Werkzeug erstellt werden, um so die Übersichtlichkeit zu wahren. Mit **SHORT-NAME** werden die einzelnen Packages identifiziert. So kann ein Entwickler ein Package für Datentypen, für Softwarekomponenten oder andere AUTOSAR-spezifischen Elemente erstellen. Dem Nutzer wird diese Hierarchie im entsprechenden Werkzeug in Form von Ordnern präsentiert, wie bspw. im Windows-Explorer.

Dieses Unterkapitel soll als Basis für die folgenden Unterkapitel dienen, da diese Anhand von XML-Listing erklärt werden. Die einzelnen Listings können als ein Teil der bereits erklärten Packges betrachtet werden, das heißt oberes Listing ist als oberste Instanz für die Folgenden Listings zu sehen.

2.1.2 Applikationsbestandteile und VFB

In diesem Kapitel wird nicht wie bei den meisten Veröffentlichungen zum Thema AUTOSAR, nur reine Theorie vermittelt, sondern direkt ein Bezug zur Praxis hergestellt. Dies soll erreicht werden, indem zu jedem Unterthema ein Code-Listing bereitgestellt wird, welches nach etwas Theorie erläutert wird.

Laut AUTOSAR Spezifikation [AUT14h] ist eine Applikation ein Model einer Komposition von untereinander verbundenen Softwarekomponenten (**SW-C**). Die Realisierung der Kommunikation übernimmt der Virtual Functional Bus (**VFB**).

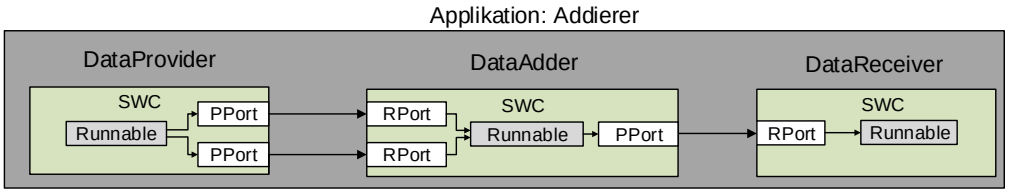

Abbildung 2.2: Aplikation: Addierer

Abbildung 2.2 zeigt die Komponenten anhand eines Beispiels. Die Applikation Addierer besteht aus drei SW-Cs, welche jeweils eine Runnable und Ports verwaltet. Diese Ansicht entspricht dem Konzept des Virtual Functional Bus, da es sich hierbei um die reine Modellierung der Anwendung handelt (siehe [Sch14]).

Softwarekomponenten und Ports

Softwarekomponenten sind wie oben erwähnt nur auf Systemebene vorhanden. Sie können als Container für weitere Elemente betrachtet werden. Die SW-Cs aus dem Beispiel von 2.2 werden als Anwendungssoftwarekomponenten bezeichnet [SL05]. Eine Spezialisierung der SW-Cs stellen die Sensor/Actuator-Komponenten dar. Diese abstrahieren die Eigenschafen von Sensoren oder Aktuatoren auf Systemebene ([AUT14i]). **Ports** sind wie in [Sch14] beschrieben, die Interaktionspunkte zwischen Softwarekomponenten und werden in zwei Arten unterschieden. Zum einen den sendenden Port, welcher als Provide Port (**PPort**) und einem Empfangsport welcher als Require Port (**RPort**) bezeichnet wird. Ein PPort stellt Daten bereit, während der RPort Daten erwartet. Außerdem bietet er die Möglichkeit Daten (Übergabeparameter) von Außerhalb an eine Runnable weiter zureichen oder ein Ergebnis nach Außen bereitzustellen. Die Eigenschaften von Ports werden durch Interfaces festgelegt. Letzteres wird im Laufe dieses Kapitels näher erläutert.

```
<APPLICATION-SOFTWARE-COMPONENT-TYPE UUID="5">
  <SHORT-NAME>AtomicSWC</SHORT-NAME>
  <PORTS>
       <R-PORT-PROTOTYPE UUID="4">
         <SHORT-NAME>In</SHORT-NAME>
              <DATA-ELEMENT-REF DEST="DATA-ELEMENT-PROTOTYPE">...
                ...
       </R-PORT-PROTOTYPE>
       <P-PORT-PROTOTYPE UUID="3">
         <SHORT-NAME>Out</SHORT-NAME>
              <DATA-ELEMENT-REF DEST="DATA-ELEMENT-PROTOTYPE">...
                ...
       </P-PORT-PROTOTYPE>
  </PORTS>
</APPLICATION-SOFTWARE-COMPONENT-TYPE>
```

Listing 2.2: SW-C Ausschnitt aus einer ARXML

Das Listing zeigt den Aufbau einer Softwarekomponente mit einem Empfangs- und einem Sende Port. Anhand dieser Informationen kann mit einem geeigneten Parser eine Softwarekomponente mit dem Namen „AtomicSWC“ und den jeweiligen Ports „In“ und „Out“ erstellt werden. Die Namen aller Elemente werden dem Tag **SHORT-NAME** entnommen. Es ist zu beachten, dass die Informationen im Listing unvollständig sind und nur als Beispiel dienen sollen. In einer realen Applikation kann ein solches Dokument Hunderte von Zeilen an Informationen zum Gesamtsystem beinhalten. Der erste ”Tagïm Listing, leitet den Beginn einer Softwarekomponenten ein. ”Ports”beschreibt eine Auflistung aller Interaktionspunkte die auf Systemebene zur übergeordneten SW-C gehören. Außerdem ist dem Listing 17 zu entnehmen, dass mit **DATA-ELEMENT-REF** eine Referenz auf den Datentyp vorhanden ist. Bei der Betrachtung des Gesamten Listings fällt auf, dass die Beschreibung, wie Ports in einer SW-C verwaltet werden, auch der strukturellen Ansicht des VFBs entsprechen.

Auf Implementationsebene wird ein Port in der entsprechenden Quellcode-Datei in der RTE definiert. Er stellt dabei eine Standard API zur Verfügung, welche vom SW-C-Namen, dem entsprechenden Port und dem dazugehörigen Datentypen abgeleitet werden. Bezogen auf das Beispiel Adder würden die Funktionen dem folgenden Listing entsprechen.

```
Std_ReturnType Rte_Read_AtomicSWC_AIn_DataElement(DataType * DataElement)

Std_ReturnType Rte_Write_AtomicSWC_COut_DataElement(DataType DataElement)
```

Listing 2.3: standardisierte Aufrufe

Laut Spezifikation [AUT14h] setzt sich ein Portkommando immer aus der Übergeordneten SW-C, dem gerade benutzen Port und dem zu übergebenden Datenelement zusammen. Handelt es sich um einen Provided Port wird die Funktion mit **Rte_write** und bei einem Receive Port mit **Rte_read** eingeleitet. Für Server-Client Operationen beginnt eine Funktionsdeklaration mit **Rte_Call**. Diese Herangehensweise erlaubt SW-C unabhängige Funktionsaufrufe, da nirgends im Funktionsaufruf auf den Zielport oder die Zielsoftwarekomponente verwiesen wird.

Runnable Entity

Eine Runnable Entity ist die eigentliche Verarbeitungslogik einer Softwarekomponente. Durch ein bestimmtes Event, wird die Runnable aufgerufen und die zugehörige Funktion ausgeführt. Ein **Event** wird z. B. durch ein Zeitliches Verhalten oder den Empfang von Daten (DataReceivedEvent) betrachtet [KF09]. Nach Abarbeitung der Code-Sequenzen, wird ein entsprechendes Ergebnis zurück geliefert. Das Resultat wird anschließend meist über den PPort bereitgestellt oder an eine weitere Interne Variable(Interrunnable Variable) übergeben. Runnable Entitys werden grundsätzlich in drei Kategorien eingeteilt.

- Cat 1A: kurze
- Cat 1B: endliche
- Cat 2: kann endlos laufen

```
<INTERNAL-BEHAVIOR UUID="674e08b0-b8cc-4e17-a947-9e27abde1399">
  <SHORT-NAME>ib_CanSender</SHORT-NAME>
  <COMPONENT-REF DEST="APPLICATION-SOFTWARE-COMPONENT-TYPE">...
  <EVENTS>
  <TIMING-EVENT UUID="7236a358-58fa-46c9-b452-ce8e29f2e597">
    <SHORT-NAME>TimingEvent</SHORT-NAME>
    <START-ON-EVENT-REF DEST="RUNNABLE-ENTITY">...
    <PERIOD>0.1</PERIOD>
  </TIMING-EVENT>
  </EVENTS>
  <RUNNABLES>
  <RUNNABLE-ENTITY UUID="46632889-2b0d-411a-a54c-4afbb2ce406c">
    <SHORT-NAME>Transmit</SHORT-NAME>
    <DATA-SEND-POINT UUID="d9afca16-72af-4c7a-86d7-99bbef4a8df7">
      <SHORT-NAME>
      <DATA-ELEMENT-IREF>
      <P-PORT-PROTOTYPE-REF DEST="P-PORT-PROTOTYPE">...
      <DATA-ELEMENT-PROTOTYPE-REF DEST="DATA-ELEMENT-PROTOTYPE">...
      </DATA-ELEMENT-IREF>
    </DATA-SEND-POINT>
    <SYMBOL>AtomicSWC_RB_adder</SYMBOL>
  </RUNNABLE-ENTITY>
  </RUNNABLES>
</INTERNAL-BEHAVIOR>
```

Listing 2.4: Runnable Ausschnitt aus einer ARXML

In der Xml Definition ist eine Runnable-Entity ein Teil des internen Verhalten, weil hier die Ausführung dieser festgelegt wird. Wie das Listing 25 zeigt, wird das interne Verhalten mit dem Start-Tag **INTERNAL-BEHAVIOR** eingeleitet. Die Softwarekomponente die das Verhalten implementiert, wird dem Tag **COMPONENT-REF** entnommen. Innerhalb des **EVENTS** Tags, werden alle Konfigurierten Events aufgelistet. Im Listing ist das beispielsweise ein **TIMING-EVENT**, das heißt ein Zeitlich immer wiederkehrendes Ereignis. Mit **START-ON-EVENT-REF** wird die Ziel Runnable festgelegt, welche alle 100 ms (**PERIOD**) aufgerufen wird. Anschließend folgt die Definition der Runnable Entities selbst (**RUNNABLE-ENTITY**). Dabei leitet der Tag **DATA-RECEIVE-POINT** die Abarbeitung der Codesequenzen ein, wenn Daten empfangen wurden. Mit **P-PORT-PROTOTYPE-REF**, wird

der Sende-Port angegeben, über welchen, die Ergebnisse der Runnable bereitgestellt werden und **DATA-ELEMENT-PROTOTYE** zeigt den zugehörigen Datentyp. Entgegengesetzt zum Daten-Sende-Punkt, wird mit DATA-RECEIVE-POINT auf ankommende Daten reagiert, diese werden aufgrund der Menge an weiteren Zeilen im Listing nicht gezeigt. Auch hierbei wird der zugehörige Empfangsport (R-PORT-PROTOTYPE-REF) mit entsprechendem Datentyp (DATA-ELEMENT-PROTOTYE) festgelegt. Zum Schluss ist noch der Tag **SYMBOL** zu erklären. Dieser beinhaltet den tatsächlichen Namen der Runnable Entity, wie er auch später als Funktion aufgerufen wird [AUT14h].

Interfaces

Die Kommunikation zwischen Softwarekomponenten setzt voraus, dass den beteiligten Ports der zu übertragende Datentyp bekannt ist. Einem Interface können mehrere zulässige Datentypen zugeordnet werden. Die AUTOSAR-Spezifikation ([AUT14h]) schreibt vor, dass zwei datenaustauschende Ports das gleiche Interface und somit die selben Datentypen zugewiesen werden müssen. Des weiteren werden Schnittstellen in zwei Arten unterschieden. Zum einen in Sender/Receiver (S/R) und zum anderen in Client/Server (C/S)([AUT14h]). Bei einem C/S-Interface Stellt ein Server Dienste bereit, welche von den entsprechenden Clients verwendet werden können. [KF09].

```
<SENDER-RECEIVER-INTERFACE UUID="9f41c51f-fd92-41b9-9303-3097e9ba8b45">
  <SHORT-NAME>EcuSts</SHORT-NAME>
  <DATA-ELEMENTS>
    <DATA-ELEMENT-PROTOTYPE UUID="9f237cb9-6d26-4623-98cd-5e57aaf7f314">
      <SHORT-NAME>IFAdder</SHORT-NAME>
      <TYPE-TREF DEST="INTEGER-TYPE">/DataTypes/tuc_uint8</TYPE-TREF>
      <IS-QUEUED>false</IS-QUEUED>
    </DATA-ELEMENT-PROTOTYPE>
  </DATA-ELEMENTS>
</SENDER-RECEIVER-INTERFACE>
<CLIENT-SERVER-INTERFACE>
  <SHORT-NAME>EcuM_ApplicationMode</SHORT-NAME>
  <OPERATION-PROTOTYPE>
    <SHORT-NAME>SelectApplicationMode</SHORT-NAME>
    <ARGUMENTS>
    <ARGUMENT-PROTOTYPE>
      <SHORT-NAME>appMode</SHORT-NAME>
      <TYPE-TREF DEST="INTEGER-TYPE">/AUTOSAR/Services/EcuM/EcuM_AppModeType</TYPE-TREF>
      <DIRECTION>IN</DIRECTION>
    </ARGUMENT-PROTOTYPE>
    </ARGUMENTS>
  ...
</CLIENT-SERVER-INTERFACE>
```

Listing 2.5: Interface Ausschnitt aus einer ARXML

Die Beschreibung wird mit dem Interface-Typ begonnen. Im Listing, ist dies ein S/R **Sender-Receiver-Inerface**. Neben dem Namen ist die Information über das Datenelement von hoher Relevanz. Letzteres wird mit dem Tag **DATA-ELEMENT-PROTOTYP** zugeordnet. Die Referenz auf den zugeordneten Datentyp ist dem Listing mit **TYPE-TREF** zu entnehmen. Eine weitere wichtige Eigenschaft die ein Interface einem Port verleiht, ist die Datensemantik. Mit **IS-QUEUED** wird festgelegt ob immer nur der zuletzt übermittelte Wert zählt (**true**) oder ob nach einem ty-

pische FIFO-Verhalten[1] (**false**) gearbeitet wird [KF09]. Beim S/R-Interface wird mit startet der Client eine sogenannte Operationen (OPERATION-PROTOTYPE). Mit dem entsprechenden Parameter (ARGUMENT-PROTOTYPE) können auch Daten übertragen werden. Die Richtung der Übertragung wird mit DIRECTION festgelegt.

Assembly Connector

Ein Assembly Connector oder einfach Connector ermöglicht die Zuordnung der Verbindungen unter den Ports auf Systemebene [KF09]. Bei einem Modelierungsprogramm für AUTOSAR-Applikationen ermöglicht die Verbindung zweier Ports mit einem Connector, den Datenaustausch. Das funktioniert aber nur wenn den entsprechenden Ports ein gemeinsames Interface zu geordnet wurde.

```
<CONNECTORS>
  <ASSEMBLY-CONNECTOR-PROTOTYPE UUID="7b...">
    <SHORT-NAME>AssemblyConnectionValue1</SHORT-NAME>
    <PROVIDER-IREF>
    <COMPONENT-PROTOTYPE-REF DEST="COMPONENT-PROTOTYPE">...
    <P-PORT-PROTOTYPE-REF DEST="P-PORT-PROTOTYPE">...
    </PROVIDER-IREF>
    <REQUESTER-IREF>
    <COMPONENT-PROTOTYPE-REF DEST="COMPONENT-PROTOTYPE">...
    <R-PORT-PROTOTYPE-REF DEST="R-PORT-PROTOTYPE">...
    </REQUESTER-IREF>
  </ASSEMBLY-CONNECTOR-PROTOTYPE>
</CONNECTORS>
```

Listing 2.6: Connector Ausschnitt aus einer ARXML

Im Listing 14 ist zu sehen, dass eine Portverbindung durch **CONNECTORS** angezeigt wird. Die relevanten Eigenschaften sind die Referenz zur Softwarekomponente (**COMPONENT-PROTOTYPE**) und zum zugehörigen PPort (**P-Port-PROTOTYPE**) oder RPort (**R-Port-PROTOTYPE**).

[1] FIFO: First In First Out. Beschreibt die Abarbeitung der Daten als Schlange

2.1.3 RTE

Bisher wurden Softwarekomponenten und Ports aus Sicht des Virtual Functionl Bus betrachtet. Die Run Time Environment realisiert hingegen die tatsächliche Kommunikation der Softwarekomponenten untereinander und mit der Basissoftware ([KT13]). Diese Schicht wird auf Quellcode-Ebene nach spezifischen Vorgaben generiert.

RTE-Generierung

Ein RTE Generator hat die Aufgabe alle benötigten Bestandteile für eine Kommunikation von Softwarekomponenten zu ermöglichen [AUT14h]. Hierbei werden Code Dateien generiert, welche das Senden und Empfangen von Daten (Funktionsaufrufe), realisieren. Des weiteren werden so getanen Taskbodys generiert, welche zur Laufzeit vom AUTOSAR-Betriebssystem aufgerufen werden können.

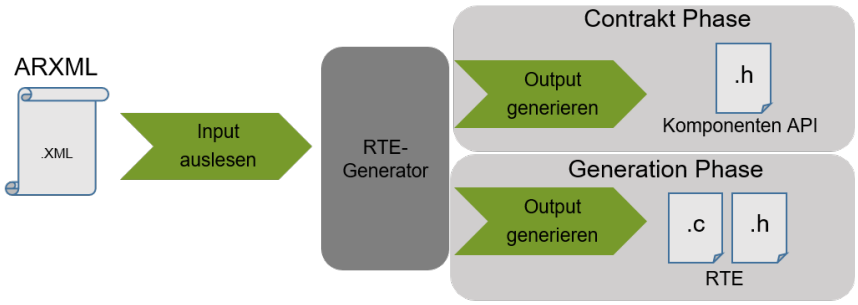

Abbildung 2.3: RTE Generierungphasen

Wie in der stark vereinfachten Abbildung 2.3 zusehen, werden die Informationen aus der ARXML (siehe 2.1.1) gelesen, da hier alle benötigten Daten zur Applikation enthalten sind. Aus diesen werden anschließend die zur Kommunikation benötigten Quellcode Dateien generiert. Dieser Vorgang wird laut Spezifikation ([AUT14h]) in zwei Phasen unterteilt.

Contract Phase In dieser Phase wird der „Vertrag“ für die Schnittstellen der Softwarekomponenten festgelegt ([AUT14h]). Nach C und C++ Standard werden hierfür die Header-Dateien erzeugt, welche die Deklaration für die Funktionsaufrufe beinhalten.

Generation Phase Wie vom Namen abzuleiten wird in dieser Phase die RTE generiert. Nach dem alle Schnittstellen in der Vorhergehenden Phase bekannt gemacht wurden, erfolgt die eigentliche Implementierung aller Funktionen und Signale. Dies geschieht streng nach Spezifikation, da hier standardisierte Schnittstellen für den Datenaustausch implementiert werden.

2.1.4 Basis Software Module

Die Entkopplung von Hardware und Anwendungssoftware wird durch die Basissoftware realisiert. Die einzelnen Funktionsmodule der Basissoftware sind unabhängig von einander und können deshalb von verschiedenen Herstellern produziert und zu einem gesamten Softwarepaket zusammengefasst werden ([KF09]). Die für diese Arbeit wichtigen Funktionsmodule sind das Betriebssystem (OS), der ECU StateManager (EcuM), der BSWModeManager, der CommunicationManager (ComM) und die ComplexDeviceDriver.

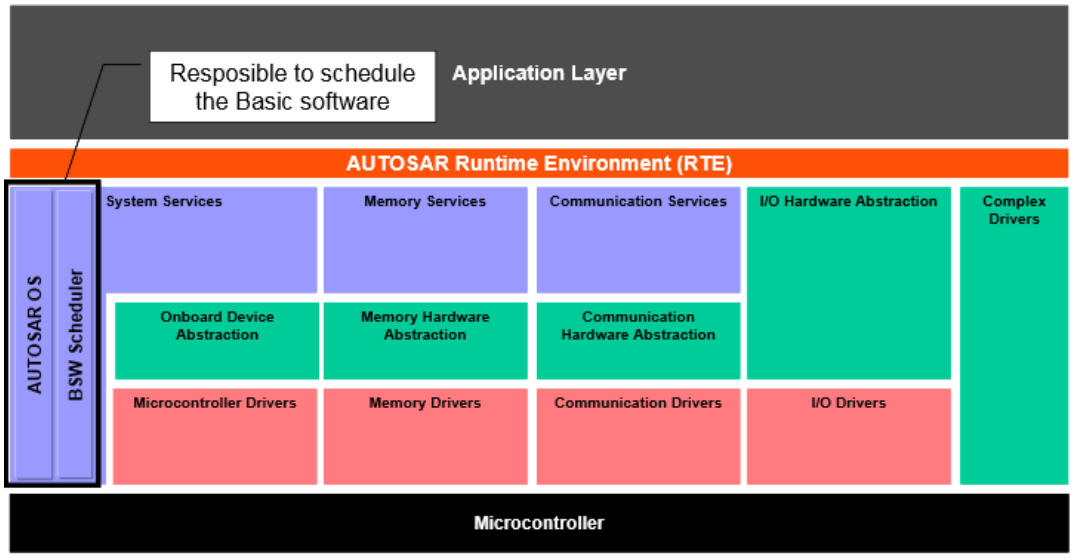

Abbildung 2.4: Basissoftwaremodule Quelle: [AUT11]

Abbildung 2.4 zeigt unter Anderem die drei Services System,Memory und Communication in der Basissoftware. Im Bereich des Systemservice ist das Betriebssystem (**AUTOSAR OS**) und das Basissoftwaremodul **BSW Scheduler** hervorgehoben. Mit dem BSW Scheduler verfügt das OS über die Möglichkeit generierte Taskbodys in der RTE und in der Basisoftware auszuführen. Das Betriebssystem ist für das Ausführen der gesamten Software zuständig. Wie am PC auch werden hierbei alle Programme in Tasks ausgeführt. Der Scheduler ist dabei für die Verwaltung der einzelnen Tasks zuständig.

ECU-Statemanager

Der **ECU Statemanager** (EcuM) verwaltet die Zustände (STARTUP, Run, SHUTDOWN, SLEEP, WAKEUP, OFF) die ein Autosar konformes Steuergerät annehmen kann [AUT14d]. Er übernimmt nicht nur die Initialisierung und das Herunterfahren der BSW-Module sondern auch vom OS und der RTE([AUT14d]).

Communication-Manager

Für die Verwaltung aller an der ECU angeschlossenen Kommunikationswege, ist der **Communication Manager** (ComM) verantwortlich. Er vereinfacht die Benutzung und Koordiniert die Verfügbarkeit vom Bus-Kommunikations-Stack ([AUT14c]).

Zentrales Glied für das Senden und Empfangen auf dem BUS (z.B. CAN) ist das **Com**-Modul ([AUT14b]). Mit Funktionsaufrufen wie `Com_SendSignal` oder `Receive_Signal`, verfügt die **RTE** über die Möglichkeit Daten auf dem Transportmedium zu lesen oder zu empfangen (siehe [AUT14c]). COM sorgt dafür dass Signale oder Signalgruppen von der RTE zum PDU-Router gesendet oder vice versa empfangen werden (siehe [HLHA13]). Der PDU Router verteilt die Pakete zu jeweiligen Transport Protokol Modulen wie z. B. CAN TP (siehe [AUT14f]). Durch die jeweiligen Identifikatoren im Header der PDUs wird eine Bussystem unabhängige Realisierung der Datenübertragung gewährleistet.

BSW-Scheduler

Der BSW scheduler ist für das Ausführen aller BSW-Module verantwortlich ([KF09]). Des weiteren übernimmt er auch die Ausführung der Runnables (2.1.2) in den Softwarekomponenten. In AUTOSAR wird eine Runnable von einem Betriebssystem Task ausgeführt. Für die Verwaltung von Tasks ist der OS Scheduler verantwortlich.

> „OSEK OS defines an event-driven priority scheduling. Tasks are assigned static priorities, which cannot be changed by the user at runtime“ [KYM+09]

Das heißt, AUTOSAR benutzt festgelegte Prioritäten für das Abarbeiten von Tasks. Das liegt unter anderem daran, das die Konfiguration des Systems ebenfalls aus der ARXML ausgelesen wird. Im Gegensatz zum dynamischen scheduling (z.B. EDF), bieten feste Prioritäten den Vorteil, dass für die CPU auf dem Steuergerät keine extra Berechnungen (Overhead) anfallen.

2.2 Testen von Software

Das Testen von Software ist in der heutigen Zeit einer der Grundlegenden Pfeiler im Lebenszyklus eines Programms. Das beginnt schon in der Frühen Entwicklungsphase bis hin zum späteren Kunden-Support. Hierfür haben sich einige Verfahren oder Modelle bewährt und stellen einen festen Standard in der Softwareindustrie dar. Ein wichtiger Begriff der, in den Folgenden Kapiteln des öfteren eine Anwendung findet ist das **Testobjekt**. Dieses beschreibt ein Objekt, eine Prozedur oder andere Softwareteileinheiten, die mit speziellen Prüfverfahren verifiziert und validiert werden. Da ein Testobjekt sowohl das ganze Programm als auch nur Teile von diesem darstellen kann, ist es erforderlich eine separate **Simulationsumgebung** zu schaffen. Letztere hat die Aufgabe das zu testende Objekt auszuführen und entsprechende Abhängigkeiten zu simulieren. Dieses Kapitel beschäftigt sich..

2.2.1 Statischer Test

Ein statischer Test hat den Vorteil, dass das zu testende Programm nicht ausgeführt werden muss und somit entfällt auch das erzeugen von Testfällen. Statische Testverfahren können in Software Reviews und statische Analyse unterschieden werden. Bei **Software Reviews** wird der Programm-Code von einer oder mehreren Personen analysiert. Bei einer **statischen Analyse** wird meist ein Werkzeug benutzt, welches den Quellcode des Testobjektes prüft([SL05]). Ein sehr bekanntest Werkzeug für statische Analysen ist der Compiler.

2.2.2 Dynamischer Test

Im Gegensatz zum statischen Test wird das Testobjekt Ausgeführt ([SL05]) und bietet somit die Möglichkeit, verschiedene Testverfahren anzuwenden. Diese werden anhand der vorliegenden Spezifikation für das jeweilige Testobjekt erstellt und somit Objektspezifische Testwerte generiert. Die Testverfahren lassen sich in **Blackbox**- und **Whitebox**- Tests unterteilen.

Blckbox-Verfahren

Hierbei handelt es sich um eine Spezifikationsorientierente Methode einen Test durchzuführen. Die Struktur und somit der interne Aufbau des Testobjektes ist nicht bekannt ([Cle10]). Es sind also lediglich die Eingangs- und Ausgangsschnittstellen relevant. In vielen Veröffentlichungen zum Thema werden Blackbox-Verfahren als Schwarze Kiste betrachtet, deren innen Leben nicht zu sehen ist. Bei der Testfallerstellung ist es nicht unüblich, dass sehr viele Kombinationen entstehen, welche es zeitlich bedingt, unrealistisch machen, alle Testfälle anzuwenden ([SL05]). Daher gibt es einige, etablierte Verfahren die im Folgenden Kurz erläutert werden.

Äquivalenzklassenbildung

Hierbei handelt es sich um die Bildung von Klassen aus Eingabe- oder Ausgabewerten. Tritt ein Fehler bei dem Wert einer bestimmten Klasse auf, wird vermutet dass die anderen Repräsentanten dieser Klasse den selben Fehler hervorruft. Somit lassen sich Klassen mit gültigen und ungültigen Werten bilden. Umgedreht wird bei Klassen, in denen ein gültiger Wert auftritt, davon ausgegangen, dass alle anderen auch gültig sind ([WEMS+12]).

Grenzwertanalyse

Den Grenzen eines Datentyps, wird in Tests zu meist keine Bedeutung geschenkt, weshalb die Überschreitung dieser immer wieder zu Fehlern führt([WEMS+12]). Bei einer Grenzwertanalyse wird die unmittelbare Umgebung eines Datentyps oder eines entsprechend festgelegten Intervalls geprüft.

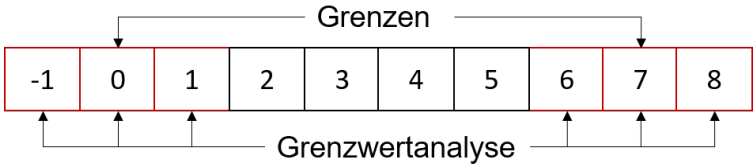

Abbildung 2.5: Grenzwertanalyse Uint8

Abbildung 2.5 zeigt den Wertebereich anhand eines Uint8 Datentyps. Die unterste und oberste **Grenze** wird durch das Feld 0 (1) und 7 (128) repräsentiert. Die Grafik zeigt außerdem die unmittelbare Umgebung dieser Grenzen(Rot). Für die **Grenzwertanalyse** werden dementsprechend die Felder -1, 0, 1 und 6, 7, 8 geprüft. Diese Analyse lässt sich aber auch mit der bereits behandelten Äquivalenzklassenbildung kombinieren. So können die Grenzwerte der gültigen Wertebereiche analysiert werden ([WEMS+12]).

Whitebox-Verfahren

Ein Whitebox-Test ermöglicht es, die einzelnen Strukturen in einem Testobjekt zu prüfen. Damit ist der logische Ablauf in einem Programm gemeint, der es ermöglicht Schleifen oder Verzweigungen gezielt mit Testwerten zu überprüfen([WEMS+12]). Mögliche Testverfahren sind z. B. ein **Kontrollflussbasierter Test**, oder ein **Datenflussbasierter Test**.

Werkzeuge für dynamische Tests

Für die Ausführung des Testobjektes, bedarf es einem Werkzeug, welches die eigentliche Umgebung simuliert und mit Testwerten speist. Zumeist auch als Simulationsumgebung oder Simulator bezeichnet, wird hiermit die Möglichkeit der automatisierten Steuerung eines Test geboten.

> „Die Werkzeuge versorgen das Testobjekt mit Eingabedaten, zeichnen die Reaktion des Testobjekts auf und protokollieren den Testlauf.“[SL05]

2.2.3 Teststufen im V-Model

Das V-Model beschreibt eine Vorgehensweise für die Entwicklung eines Softwareprojekts. Dabei decken die einzelnen Phasen den gesamten Lebenszyklus der Software ab. Mit seinen zwei V-Förmigen Ästen wird die Entwicklung und das Testen der Software parallel und somit gleich priorisierend vorgeschrieben ([SL05]). Da sich diese Arbeit nur mit dem Testen von Software beschäftigt, ist auch nur der rechte Ast des V-Models von Bedeutung. Vielmehr nur die zwei Stufen Komponenten- und Systemtest, welche im Folgenden kurz erläutert werden. Alles Testarten sind Teil des dynamsichen Testens, da hier eine Ausführung des zu testenden Obejktes vorliegt.

Komponententest

Komponententests stellen eine Überprüfung der kleinsten Softwareeinheiten im System dar (siehe [SL05]). Abhängig von der jeweiligen Programmiersprache, wird das Testen einzelner Komponenten beispielsweise auch als Unit-, Klassen-, oder Modultest bezeichnet. Hierbei wird die zu testende Komponente als isoliertes Testobjekt eines Gesamtsystems betrachtet. Stimuli von anderen Einheiten haben somit keine Relevanz mehr. Bevor eine einzelne Software-Komponente ihre Dienste im Hauptprojekt verrichten kann, muss sie folglich getestet werden. Hierbei ist es sehr wichtig dass das Software-Modul nicht nur verifiziert sondern auch Validiert wird.

> „Wichtigste Aufgabe des Komponententests ist die Sicherstellung, dass das jeweilige Testobjekt die laut seiner Spezifikation geforderte Funktionalität korrekt und vollständig realisiert..“ [SL05]

Um die Komponenten auf ihre spezifischen Anforderungen zu testen, gilt es gezielt Testfälle zu erstellen, welche die entscheidenden funktionalen Anteile überprüfen. Im Gegensatz zu anderen Testverfahren, ist es bei Unit-Tests ehr einfacher, automatisierte Testverläufe zu erstellen([Cle10]). Mit Hilfe von Testrahmen, können Komponenten vom Gesamtsystem isoliert werden.

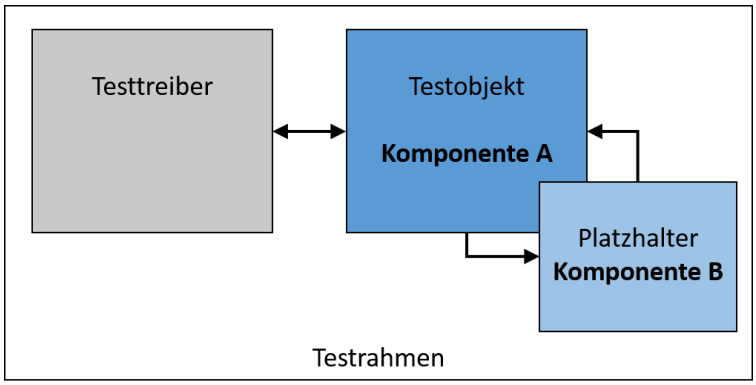

Abbildung 2.6: Testrahmen mit Testreiber und Platzhalter nach [Cle10]

Ein **Testrahmen** besteht aus einem Testtreiber und einem Platzhalter. Der **Testtreiber** ist die eigentliche Ausführungseinheit für eine Komponente. Die Isolierung vom System ermöglicht der **Platzhalter**. Dazu übernimmt er die Rollen von den vom Testobjekt abhängigen Komponenten. Hierbei wird zwischen drei Arten unterschieden ([Cle10]).

- stub
- dummy
- mock

Ein **stub** ist ein Platzhalter der eine oder mehrere benötigte Funktionen für das Testobjekt bereitstellt. Hierbei implementiert ein **realistischer** stub einen Großteil der Funktionalität und ein **spezifischer** nur die benötigten Funktionen zum Ausführen der Komponente ([WEMS+12]). Der **dummy** realisiert eine vollständige Attrappe der abhängigen Komponenten und wird, sobald die Funktionalität dieser zum Einsatz kommt, auch als **mock** bezeichnet([Cle10]).

Systemtest

Das Testen aller Softwareeinheiten im Verbund wird als Systemtest bezeichnet. Hierbei werden die Schnittstellen zur Außenwelt mit Testwerten stimuliert oder im Falle des Outputs abgehört. Diese Art von Test kann ebenfalls als Blackboxtest bezeichnet werden, da die inneren Strukturen hierbei nicht ersichtlich sind. Es Zählt lediglich der In- und Output. Auch für das Gesamtsystem werden spezielle Testfälle erzeugt, die die spezifischen Anforderungen abdecken.

2.3 Zusammenfassung

Das Grundlagenkapitel hat die Erläuterung aller für diese Arbeit relevanten Themen Aufgezeigt. Dabei wurden zuerst elementare Grundkenntnisse von AUTOSAR Vermittelt. Mit Hilfe entsprechender Code-Listings wurde direkt ein praktischer Bezug zu Softwarekomponenten, Interfaces, Datentypen und deren Verarbeitung in einer AUTOSAR spezifischen XML-Datei hergestellt. Dies ermöglicht dem Lesenden einen schnellen Einstieg in das Thema ARXML-Parsen, da die wichtigsten Tags erklärt wurden.
Das Thema Softwaretests ist ebenfalls eine tragende Säule für diese Arbeit. Dazu wurde aufgezeigt, dass zwischen statischem und dynamischen Prüfverfahren unterschieden wird. Bei ersterem ist es nicht erforderlich das Testobjekt auszuführen und mit Testwerten zu prüfen. Bei einem dynamischen Testverfahren hingegen, wird der zu testende Prüfling zum Laufen gebracht und ermöglicht so eine Validierung und Verifizierung. Weiterhin wird die Betrachtung eines Testobjektes durch Blackbox- und Whitebox- Verfahren unterschieden. Während bei letzteren eine strukturelle Überprüfung möglich ist, werden bei zweiten nur die äußeren Schnittstellen geprüft. Die erforderlichen Testwerte werden mit Verfahren wie Äquivalenzklassenbildung und Grenzwertanalyse ermittelt. Der zweite große Abschnitt im Softwaretest Kapitel wird durch die im V-Model anzutreffenden Teststufen, Komponent- und Systemtest abgerundet.

3 Stand der Technik

Dieses Kapitel beschäftigt sich mit den derzeitigen Möglichkeiten Applikationen in AUTOSAR zu testen. Dabei werden kurz mögliche Testansätze erläutert und diese anschließend nach Kriterien geordnet. Außerdem werden AUTOSAR-Testprogramme vorgestellt, die festgelegten Testkriterien zugeordnet werden können.

3.1 Aktuelle Testansätze

Für das Testen von AUTOSAR-Anwendungen gibt es auf dem Markt einige Lösungen. Dabei nutzen die Hersteller verschiedene Ansätze um Steuergerätecode hardwareunabhängig in eine Simulation einzubinden. Das Ziel ist es, Fehler immer früher aufzudecken in dem die Schichten im AUTOSAR-Model einzeln betrachtet werden. Bei dieser Arbeit handelt es sich um das gezielte testen der Applikation. Dazu gibt es an der Professur Technische Informatik und dem dort befindlichen „Yellow Car“ Demonstrator in diese Richtung gehenden Ansätze. So wurde in der Masterarbeit mit dem Thema „Konzept für automatisierten virtuellen Test von AUTOSAR Applikationen“, die Absicherung der Lichtsteuerungsapplikation des Yellow Car's realisiert. Für die Simulation wurde VEOS (dSpace) und für den automatisierten Test ECU-TEST (TraceTronic) verwendet. Ebenfalls zu erwähnen ist die Arbeit „Testautomatisierung für funktionale Softwarekomponenten im AUTOSAR-Entwicklungsprozess“, in der es darum geht automatisierte Funktionstests für AUTOSAR-Softwarekomponenten zu erstellen und auszuführen. Die Teststufen umfassen hierbei den MIL- ,HIL- und SIL-Test. Grundbaustein ist hierbei immer die AUTOSAR XML(ARXML), welche im Grundlagenkapitel 2.1.1 beschrieben wurde. Ganz nach AUTOSAR-Spezifikation wird Anhand der XML-Datei eine RTE generiert, welche dann entsprechend alle Schnittstellen für eine erfolgreiche Kommunikation innerhalb der Simulation bereitstellt.

Für eine standardisierte Validierung von bereits konfigurierten AUTOSAR-Anwendungen bietet AUTOSAR seit 2014 die **Acceptance Tests**.

> „Sie zielen auf der Spezifikationsebene darauf ab, das Verhalten einer konfigurierten AUTOSAR-Implementierung an seinen Schnittstellen zu validieren.([aut14a])“

Das erste Release beschreibt Testfälle für die RTE, Basissoftware und die Buskommunikation. Mit der Version 1.1 kommt die Testfallspezifizierung zu Ethernet hinzu. Genaueres ist der Überersicht zu den Acceptance Tests [AUTnd] zu entnehmen.

3.2 Programm-Recherche

In diesem Kapitel werden die recherchierten Programme analysiert und auf eine Einbezugnahme ins Projekt begutachtet. Dabei wird der Fokus stark auf hardwareunabhängige- und auf reine Applikationstests gelegt. Alle Programme im Betrachtungsfeld sind **AUTOSAR-spezifisch**, das heißt sie sind in der Lage den generierten Quell-Code zu simulieren.

VEOS

VEOS ist ein Produkt von **dSpace**. Hierbei handelt es sich um eine PC-Basierte simulations Plattform, die neben **Simulink-** und **TarketLink-**Modellen auch AUTOSAR Software simulieren kann. Als Teil des **MathWorks Connection Programs** wird mit TarketLink eine optimale C-Code Generierung und Integration in die Simulationsumgebung ermöglicht. Softwaretests können in Verbindung mit SystemDesk ausgeführt werden.

DaVinci Component Tester

Der Component Tester ermöglicht ebenfalls das Testen einzelner Softwarekomponenten oder auch SWC-Komposition. Mit Hilfe eines externen Programms wie NUnit oder CANoe können verschiedene Testszenarien gesteuert und protokolliert werden. Die RTE wird anhand der Informationen der entsprechenden ARXML emuliert.

ARUnit

Artop ist ein Projekt welches 2008 von einer Gruppe von Nutzern mit zugehörigen Lizenzen gegründet wurde. Hierbei ging es den Beteiligten darum, eine gemeinsame Entwicklungsplattform auf der Basis von Eclipse zu schaffen, welche das entwickeln von Grundfunktionen für AUTOSAR-Tools bietet [KF09]. Mit Eclipse wird eine Entwicklungs- und Testumgebung für AUTOSAR-Softwarekomponenten geboten, wobei ARUnit selbst als RTE-Generator verstanden werden kann, welcher als PlugIn für Eclipse bereit steht. Es unterstützt alle AUTOSAR-Versionen und kann einfach in die Eclipse-Umgebung integriert werden. Somit erhält der Entwickler die Möglichkeit eigene Unit-Tests in Java zu programmieren. Wie in Abbildung 3.1 zusehen, verwendet ARUnit die RTE für speziell eine zu testende SW-C. Diese wird anhand der SoftWareCommponentDescription (SWCD) links im Bild generiert. Das bedeutet im Gegensatz zu normalen RTE Generatoren werden hierbei lediglich die Schnittstellen für eine zu testende SWC generiert. Dies ermöglicht einen einfachen Unit-Test. Durch die Frei zugängliche API, kann der Status der RTE überwacht und stimuliert werden. Dieser einfache Unit-Test einer Softwarekomponente hat den

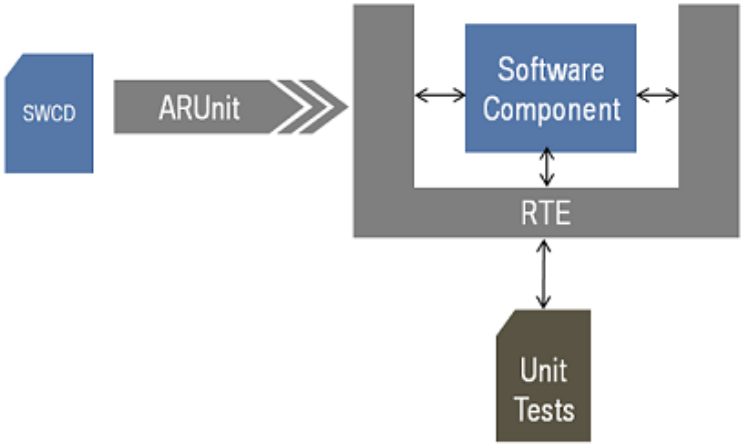

Abbildung 3.1: Arbeitsweise ARUnit

großen Nachteil, dass hierbei kein scheduling von Tasks unterstützt wird. Somit muss die ausführung der entsprechenden Runnables selbst programmiert werden.

MESSINA

Die Firma Berner und Mattner stellen mit MESSINA ebenfalls ein Testprogramm für AUTOSAR-Softwarekomponenten bereit. Auch hier ist es möglich Komponenten einzeln oder im Verbund zu testen. Die Kommunikation erfolgt im Gegensatz zu den anderen Tools nicht mit einer eigens generierten RTE, sondern mit Hilfe einer Technik namens Signalpool. D.h. die Softwarekomponenten werden unter einander Verbunden.

TPT

Mit dem modellbasierten Testwerkzeug TPT (Time Partition Testing) bietet die Firma PICTEC ein Werkzeug welches vornehmlich für das Testen von Regelungs- und Steuerungssysteme verwendet wird. Testfälle werden bei TPT mit Ablaufautomaten und Sequenzbeschreibungen modelliert. Dies vereinfacht das Verständnis eines Testfalls. Durch die Reaktivität der Automaten, können Testläufe abhängig vom zu messenden System realisiert werden. Weiterhin besteht die Möglichkeit alle Testfälle mit einem Automaten zu modellieren. Für das Testen von AUTOSAR-Komponenten ist TPT allerdings auf andere Programme wie SystemDesk, Davinci Component Tester oder Messina angewiesen. Hierzu bringt es eine Virtuelle Maschine mit, die mit den jeweiligen Softwarekomponenten als eine Simulationsumgebung bereit gestellt wird. D.h. TPT generiert unter anderem die RTE mit Hilfe der zugehörigen ARXML-

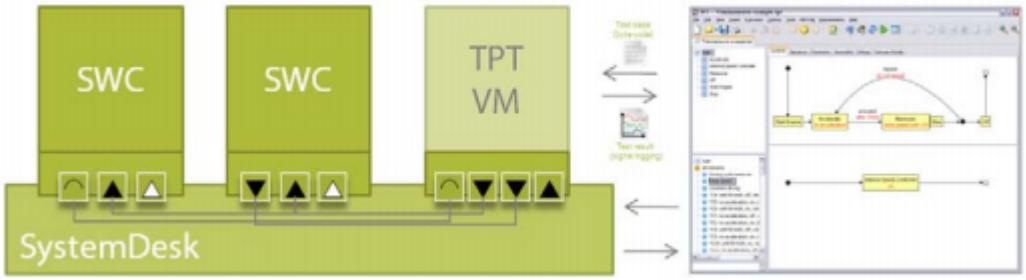

Abbildung 3.2: TPT und SystemDesk

Dateien. Obere Abbildung zeigt die Integration mit SystemDesk. TPT steuert SystemDesk fern und realisiert die Simulation und Steuerung der Testumgebung.

ISOLAR-EVE

Die Firma **ETAS** bietet mit der ISOLAR-Produktfamilie eine Lösung für die Anforderungen bei der Entwicklung und Validierung von eingebetten Systemen. AUTOSAR-Unterstützung bieten hierbei die Programme **ISOLAR-A** und **ISOLAR-EVE**. Während ISOLAR-A für die Entwicklung der Anwendungssoftware dient, wird mit ISOLAR-EVE eine virtuelle Steuergeräteplattform für die Entwicklung und den Test von AUTOSAR-Anwendungen bereitgestellt. Für die Testgenerierung wird die ARXML und der Steuergeräte Programmcode importiert [ETA15].

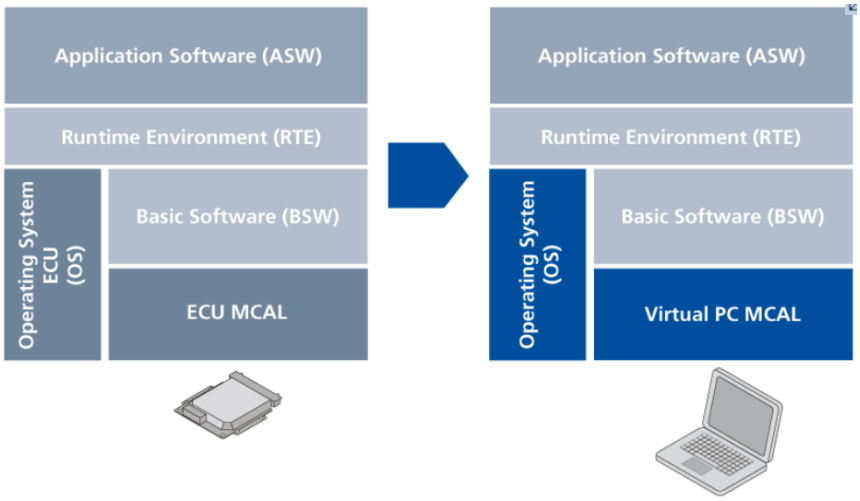

Abbildung 3.3: virtuelle Ausführung auf dem PC Quelle: [ETA15]

Wie in obiger Abbildung zusehen, wird durch die Ersetzung des Micro-Controller-Abstruction-Layers (MCAL) und des Betriebssystems in der Basissoftware, ein virtuelles Steuergerät erzeugt. Da die Ausführung auf einem Standard PC nicht in Echtzeit statt findet wird mit **RTPC-EVE** auch dafür eine Lösung bereitgestellt. RTPC-EVE dient als Simulationsumgebung für das Virtuelle Steuergerät und wird auf einem separaten PC unter RT-Linux ausgeführt [ETA15].

3.3 Auswahlkriterien der Testprogramme

Die Folgenden Kriterien beschreiben die Eigenschaften, die erfüllt sein müssen, um eine mögliche Verwendung einer Simulationslösung für diese Arbeit in Betracht zu ziehen. Die recherchierten Programme werden auf die Anforderungskriterien überprüft, indem deren Kernkompetenzen im zugehörigen Kriterium untersucht werden.

3.3.1 Hardwareunabhängiges Testen

Das **Hardware-unabhängige Testen** ist ein wesentlicher Bestandteil dieser Arbeit, da die Anforderung, dass der Test auf einem Standard-PC ausgeführt werden soll, somit grundlegend erfüllt wird. Hierbei wird die Hardware, in diesem Fall die ECU, durch die Emulierung der Hardware (z. B. V-ECU von VEOS) virtualisiert. Eine andere Herangehensweise, ist die Kompilierung des Codes in eine Ausführbare Datei. Somit übernimmt das Betriebssystem (Windows) des PC's die eigentliche Aufgabe der ECU. Es ist allerdings zu beachten, dass diese Herangehensweise nicht dem zeitlichen Verhalten auf der Zielhardware entspricht, da es sich bei den Steuergeräten um Echtzeit-Systeme handelt. Wie in Abbildung 3.1 aufgeführt, erfüllen alle Programme im Feld diese Anforderung.

3.3.2 Applikationstest

Dieses Kriterium verlangt die Abspaltung der Applikation vom Rest des Modells (siehe 2.1 AUTOSAR). Dies bezieht sich nicht nur auf die Applikationsschicht, sondern verlangt auch den Anteil der RTE, der die Kommunikation der Softwarekomponenten ermöglicht. Hierbei sollte auch das Testen der Softwarekomponenten einzeln und als Komposition möglich sein.

3.3.3 Test mit generierter RTE

Die Anforderung, dass Applikationen mit zugehöriger Run Time Environment getestet werden sollen, wird von Messina überhaupt nicht erfüllt, denn für die Kommunikation werden die Softwarekomponenten untereinander verbunden, was so auf

keiner ECU zum tragen kommt. ISOLAR-EVE ermöglicht den Import vom gesamten Steuergerätecode einer Anwendung. Das bedeutet zwar dass es als einziges Programm im Testfeld die „Fremd-RTE“ annimmt, aber auch den Rest vom AUTOSAR-Schichtenmodel. Die übrigen Programme realisieren die Tests zwar mit RTE, müssen diese aber immer generieren.

3.3.4 Test mit originaler RTE

Im vorhergehenden Abschnitt wurde erwähnt, dass die meisten untersuchten Programme die RTE selber generieren. Dieses Kriterium verlangt aber die korrespondierende RTE zur entsprechenden ARXML. Bei Betrachtung der Tabelle 3.1, ist festzustellen dass keins der untersuchten Programme alle Anforderungen erfüllt.

3.4 Zusammenfassung

Die Suche nach einem geeigneten Test-Programm für AUTOSAR-Applikationen hat gezeigt, dass manche Hersteller komplette Tool-Ketten, für jegliche Testszenarien bereitstellen. Die untersuchten Programme sind in der Lage, Softwarekomponenten einzeln oder im Verbund zu testen, was eine der Anforderungen für diese Arbeit erfüllt.

Programme	AUTOSAR-spezifische Tests	Hardware-unabhängig	Applikations-Test	Test mit generierter RTE	Mit originaler RTE
VEOS	X	X		X	
ISOLAR-Eve	X	X			X
Messina	X	X	X		
ARUnit	X	X	X	X	
TPT	X	X	X	X	
C-Tester	X	X	X	X	

Tabelle 3.1: Programm- und Kriterienübersicht

Die obere Tabelle zeigt noch einmal eine Übersicht aller Test-Programme und die für das Projekt relevanten Kriterien.

4 Konzept zur Entwicklung eines AUTOSAR-Applikations-Simulators

Die vorangegangenen Kapitel haben das nötige Grundlagenverständnis für diese Arbeit bereitgestellt. In diesem Kapitel geht es um das konzeptionelle Vorgehen für eine mögliche Implementierung. Dabei wird ein möglicher Weg aufgeführt, eine Testumgebung für AUTOSAR-Applikationen zu generieren und auszuführen.

Zielsetzung

Ziel dieser Arbeit ist es, eine Umgebung zu schaffen, die es ermöglicht AUTOSAR-Applikationen zu testen. Als statische Vorprüfung müssen alle Informationen, die die Applikation betreffen, auf Spezifikationsvorgaben und Vollständigkeit geprüft werden. Anschließend soll es möglich sein, sowohl die Applikation im Ganzen als auch einzelne Softwarekomponenten zu testen. Im Kapitel 3 Stand der Technik wurde deutlich, dass die Testprogramme auf dem Markt meist eine eigene RTE generieren, um einen Applikationstest zu realisieren. In dieser Arbeit hingegen wird die originale RTE der Anwendung in den Test miteinbezogen. Des Weiteren sollen dem Benutzer alle relevanten Systeminformationen, welche das Testobjekt betreffen, angezeigt werden. Hierfür wird ein Programm benötigt, welches in der Lage ist, die genannten Anforderungen abzudecken.

Problemstellung

Normalerweise besteht die Steuergerätesoftware aus den Softwarebestandteilen des AUTOSAR-Schichten-Modells, wie im Grundlagenkapitel 2.1 beschrieben. Das Problem besteht daher darin, die Applikation vom Rest des Modells zu trennen. Die Softwarekomponenten können ohne die Run-Time-Environment nicht untereinander oder mit der Basissoftware kommunizieren. Deswegen ist es erforderlich, den Anteil der RTE, welcher die Schnittstellen für das Austauschen von Daten bereitstellt mit als Testobjekt (siehe 2.2) zu betrachten und in die Testumgebung zu integrieren. Bei diesem Vorgehen ist zu beachten, dass es Abhängigkeiten zur Basissoftware (z. B. Funktionsaufrufe) gibt und dass der Basissoftware-Scheduler (2.1.4 Basis Software Module) nicht mehr zur Verfügung steht. Weiterhin muss ein Weg gefunden werden, die Testdaten an die Ports der SWCs zu übergeben und abzuhören. Zum Schluss muss die gesamte Applikation kompilierbar sein. D. h. es muss eine ausführbare Datei vorliegen, welche das Testen der Anwendung realisiert.

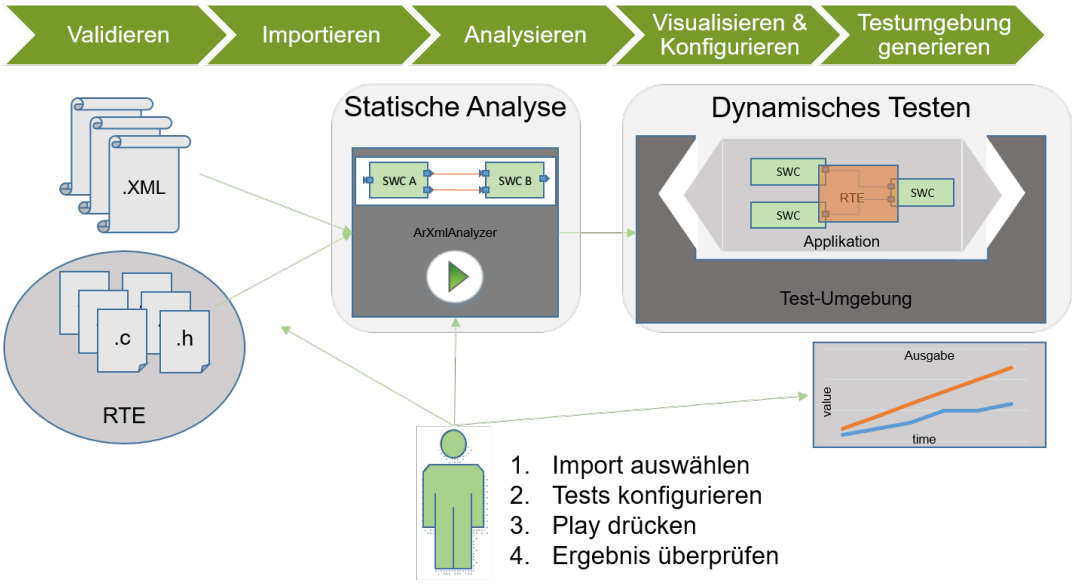

Abbildung 4.1: Projektablauf

Obige Abbildung zeigt den kompletten Ablauf bis zur fertigen Simulationsumgebung für AUTOSAR-Applikationen und anschließender Testausführung. Des Weiteren ist zu sehen dass hierbei in statisches Analysieren und dynamisches Testen Unterschieden wird. Das Testen einer Applikation besteht in dieser Arbeit, wie in Abbildung 4.1 zu sehen, aus fünf wesentlichen Schritten. Nach der Auswahl des entsprechenden Inputs über einen Dialog wird die ARXML-Datei mit dem korrespondierenden Schema validiert und auf XML-Konformität geprüft. Ist die Validierung erfolgreich, kann die Informationsgewinnung aus der/den ARXML-Datei/en gestartet werden. Sind alle Informationen gesammelt, werden diese auf AUTOSAR-Spezifikation geprüft und anschließend dem Benutzer angezeigt. Im weiteren Verlauf wird die Möglichkeit geboten, RTE-Dateien zuzuweisen, das Stimulieren und Abhören der Ports zu konfigurieren und die Testdatensätze zu erzeugen. Sind alle Informationen bereitgestellt, wird die Testumgebung generiert. Schließlich wird die Applikation im Simulator ausgeführt und mit den zuvor erzeugten Testwerten gespeist. Wichtig ist hierbei auch die Automatisierung des Tests. Das heißt, der Benutzer wählt lediglich die ARXML und die korrespondierende RTE aus und erstellt anschließend die Testfälle. Die verbleibenden Aktionen werden durch Drücken eines „Play-Buttons“ komplett automatisiert ausgeführt. Am Ende wird dem Nutzer das Ergebnis in Form eines Zeit-Wert Diagramms präsentiert.

Die folgenden Unterkapitel geben einen detaillierten Überblick über die einzelnen in Abbildung 4.1 aufgeführten Ablaufphasen.

4.1 Validieren

Bevor die ARXML geparst wird, muss das XML-Dokument auf Wohlgeformtheit geprüft werden. Die Erläuterung und die hierfür nötigen Schritte sind dem Grundlagenkapitel 2.1.1 Wohlgeformte XML Dokumente auf Seite 4 zu entnehmen.

Die Struktur und der Inhalt einer ARXML-Datei unterliegt der Spezifikation von AUTOSAR. Nur diese Herangehensweise erlaubt die Realisierung einer Applikation mit den Programmen verschiedener Hersteller. Damit diese Vorgaben eingehalten werden, wird das XML-Dokument einer Schemaprüfung unterzogen, wie im Kapitel 2.1.1 Schemaprüfung beschrieben. Die Schema.xsd kann in der entsprechenden Version von AUTOSAR.org heruntergeladen werden.

4.2 Datenimport

Das Importieren der Informationen aus einer ARXML-Datei wird durch einen XML-Parser realisiert. Der Parser muss zwei grundlegende Anforderungen erfüllen. Die Strukturierung einer AUTOSAR-Xml ist versionsabhängig und muss automatisiert angepasst werden. Zum einen können die XML-**Tags** in unterschiedlichen Ebenen vorhanden sein und weiterhin ändern sich die **Tag**-Namen selbst.

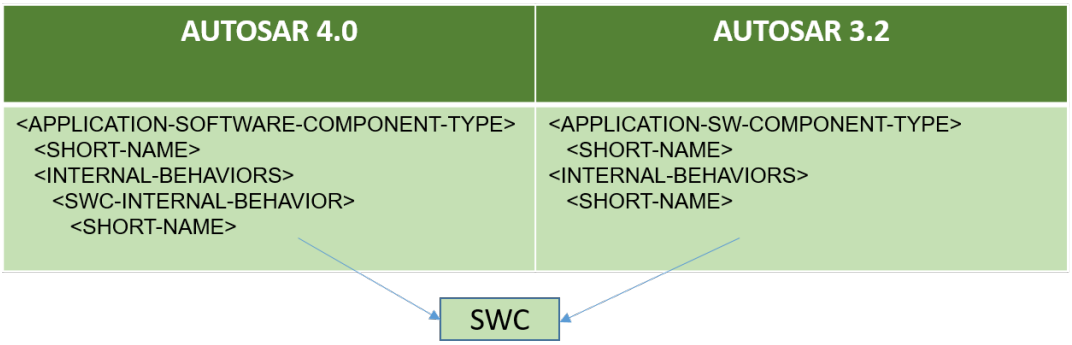

Abbildung 4.2: ARXMLVergleich

In der oberen Abbildung wird verdeutlicht, wie die Unterschiede zwischen den verschiedenen AUTOSAR-Versionen aussehen können. Der Tag für die Softwarekomponente zeigt in AUTOSAR Version 3.2 eine andere Schreibweise als in Version 4.0. Trotzdem repräsentieren beide die gleiche SWC. Sicherlich wäre es einfach die Tag-Namen für jede Version zu erstellen. Dafür müsste der Autor des Parsers aber immer selbst Hand anlegen. Damit auch später auf verschiedenste Versionen reagiert werden kann, wird hierfür ein Profilmanager bereitgestellt. Dieser bietet die Möglichkeit, für jede Version von AUTOSAR ein Profil anzulegen oder ein bestehendes zu editieren.

Diese Profile können zur Laufzeit ausgewählt und anschließend nach den entsprechenden Tag-Namen geparst werden. Somit ist es theoretisch möglich, Komponenten aus verschiedenen AUTOSAR-Versionen in einer Anwendung laufen zu lassen. Der Manager bietet hierfür eine grafische Benutzeroberfläche, welche das Editieren der Tag-Namen ermöglicht. Alternativ können XML-Profildateien importiert werden. Dadurch können Profile auch ohne das Vorhandensein des Profilmanagers von anderen Akteuren erstellt werden. Ein XML-Profil wird so einfach wie möglich strukturiert, somit soll gewährleistet werden, dass ein gewöhnlicher Text- oder XML-Editor zum Editieren verwendet werden kann. Die Suche nach Tags in anderen Ebenen kann durch einen einfachen rekursiven Algorithmus gelöst werden, der die Knoten so lang traversiert, bis die gewünschte Information gefunden wurde.

4.2.1 Objektgenerierung

Das Prüfen der Daten auf Spezifikationsvorgaben benötigt eine effiziente Methode, die die geparsten Informationen für jegliche Operationen im Speicher hält.

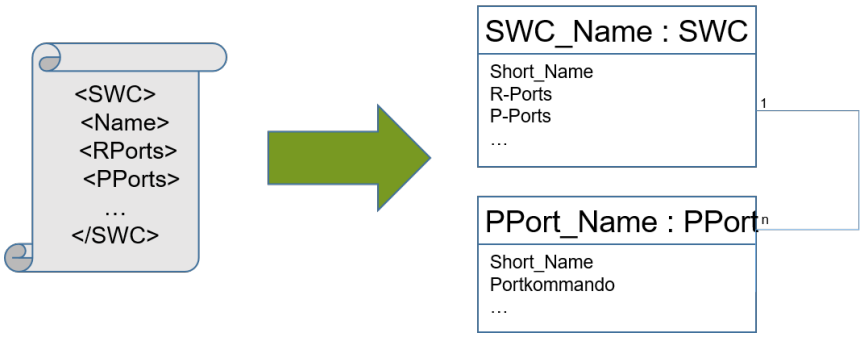

Abbildung 4.3: Objektgenerieung

Beim Lesen aus der XML-Datei werden die Informationen direkt zu Objekten überführt. Diese liegen im Speicher und sind jederzeit adressierbar. Die Abbildung 4.3 zeigt anhand einer Softwarekomponente und deren Attribute den entsprechenden Objektaufbau. Die **SWC** wird anhand des referenzierenden Tags (siehe 2.1.2) ausgelesen und als Objekt erstellt. Diese beinhaltet weitere Eigenschaften wie z. B. **Short_Name** oder weitere Unterobjekte wie **P-Port**, welches wiederum eigene Attribute besitzt. Somit ist es möglich, die Spezifikationsvorgaben anhand des Objektes auf Vollständigkeit zu prüfen. So müssen nach AUTOSAR-Spezifikation [AUT14h] beispielsweise alle Ports zur Konfigurationszeit bekannt sein.

4.3 Datenanalyse

Bei der Datenanalyse geht es um das Prüfen aller applikations-relevanter Informationen und mit welcher Methodik diese realisiert wird. Im Grundlagenkapitel 2.2.1 Statischer Test wurde bereits die statische Analyse erläutert, die zumeist mit Hilfe eines Werkzeugs realisiert wird. Für einen reibungslosen Simulationsablauf müssen sowohl die Informationen aus den ARXML-Dokumenten als auch die RTE-Dateien überprüft werden. Daher zeigen die folgenden Unterkapitel die einzelnen Anforderungen der statischen Analyse, die erfüllt werden müssen, um eine fehlerfreie Simulation durchführen zu können.

4.3.1 Objektanalyse

Als wichtige Vorprüfung in dieser Arbeit müssen alle generierten Objekte (siehe 4.2.1) auf AUTOSAR-Spezifikationsvorgaben geprüft werden. Das heißt, vor einer Simulation ist es erforderlich, dass die Attribute aller Objekte referenziert oder initialisiert sind. Hierbei können einfach die Verweise der Objekte geprüft werden. Mit einer Werkzeug-gestützten Analyse können solche Szenarien schnell und effektiv realisiert werden. Voraussetzung dafür ist das Wissen, welche Komponenten und Informationen zu prüfen sind. Eine mögliche Lösung zeigt der folgende Unterabschnitt.

Zu prüfende Informationen

Für eine Überprüfung gilt es, viele mögliche Informationen zu beachten. Die wichtigsten können den jeweiligen AUTOSAR-Spezifikationen entnommen werden. Als Leitfaden dienen für diese Arbeit die einzelnen Hinweise in den Spezifikationen. Diese beginnen immer mit dem Spezifikations-Namen, gefolgt von einer Nummerierung (bspw. VFB:001). Danach kommt die entsprechende Anforderung. Somit kann für jede zu prüfende Komponente eine Regel für das spätere Werkzeug zur statischen Analyse erstellt werden.

Interface Im Grundlagenkapitel wurde bereits erwähnt, dass z. B. miteinander kommunizierende Ports das selbe Interface aufweisen müssen, um einen einheitlichen Datentyp zu garantieren. Dazu muss lediglich überprüft werden, ob das referenzierende Interface der beiden Ports das selbe ist. Für das eben beschriebene Szeario können folgende Aussagen berücksichtigt werden:

> „VFB003: At configuration time, each port is typed by exactly one port-interface“ [[AUT14i]].
> „VFB006: At configuration time, it is known which data-elements a sender-receiver interface contains“ [[AUT14i]].

Ports Eine weitere Überprüfung zielt darauf ab, welchen Type der Port zur Konfigurationszeit aufweist.

> „VFB007: At configuration time, it is known whether a component's port is a PPort or an RPort“ [[AUT14i]].

Runnables Auch bei Runnables ist es wichtig zu wissen, welche Softwarekomponente welche Runnable aufweist.

> „VFB043: At configuration time, the runnables of a component must be known“ [[AUT14i]].

SWCs Für Softwarekomponenten müssen zur Konfigurationszeit alle Ports bekannt sein. Außerdem dürfen diese nur über ihre Ports miteinander kommunizieren.

> „VFB001: At configuration time, the component's ports are known“ [[AUT14i]].
> „VFB002: Components interact with each other through their ports only“ [[AUT14i]].

Wie in 4.2.1 Objektgenerierung beschrieben, kennt ein SWC-Objekt alle seine Ports. Daher ist diese Überprüfung schnell zu realisieren. Außerdem können in diesem Zug auch sofort die Anforderungen der Ports selber überprüft werden. Gleiches gilt auch für die Runnables.

Assembly-connector Das Grundlagenkapitel hat aufgezeigt, dass ein Connector für die Verbindung der Ports auf Systemebene verantwortlich ist. Bei falschen oder nicht vorhandenen Verbindungen können im VFB Fehler auftreten. Hierbei sind folgende Spezifikationsvorgaben zu beachten:

> „VFB010: An assembly-connector connects exactly one PPort with exactly one RPort“ [[AUT14i]].

Weiterhin muß beachtet werden, dass exakt ein PPort mit einem RPort verbunden werden kann. Und das nur, wenn die Interfaces und Typen der Ports bekannt und kompatibel zueinander sind.

4.3.2 RTE- und ARXML-Vergleich

Beim Import von mehreren Dateien kann es vorkommen, dass die RTE-Dateien nicht zur korrespondierenden ARXML gehören oder umgekehrt. Dadurch würden viele Fehler auftreten, da die Informationen der RTE und der AUTOSAR XML voneinander abhängig sind. Es muss folglich eine Methode entwickelt werden, die den Inhalt von Code-Dateien und der ARXML vergleichen kann.

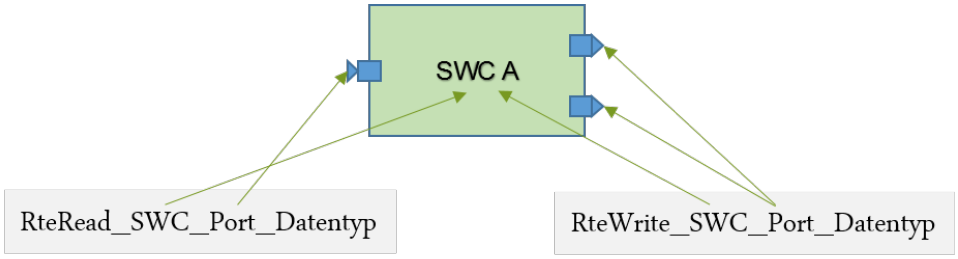

Abbildung 4.4: Nachbildung einer SWC anhand der Portkommandos

Eine mögliche Lösung für diese Anforderung findet sich in den Port-Kommandos wieder, welche in den entsprechenden RTE-Dateien vom RTE-Generator erzeugt wurden. Wie in Abbildung 4.4 zu sehen, kann anhand aller Funktionsaufrufe die von einer Softwarekomponente möglich sind, ein Rückschluss auf diese getroffen werden. Das Port-Kommando lässt sich in vier Teile aufsplitten. **Rte_Read** wird für einen Empfangsport und **Rte_Write** für einen Sendeport verwendet. Dabei ist zu beachten, dass es auch abgewandelte Varianten gibt, wie Rte_IWrite und Rte_IRead für das implizierte Schreiben und Lesen eines Puffers. Beim zweiten Teil eines Kommandos handelt es sich um den Namen der Softwarekomponenten und beim dritten Teil um den Namen des Ports. Den Abschluss bildet die Bezeichnung des Datentyps. Die gewonnenen Informationen können so mit den im Speicher befindlichen Objekten (siehe 4.2.1) abgeglichen werden. Können alle SWCs über die RTE-Funktionsaufrufe rekonstruiert werden, ist davon auszugehen, dass es sich um die Dateien des selben Projektes handelt. So müsste Beispielsweise eine Softwarekomponente mit dem Namen „SWC“, den zugehörigen RPort „In“ und PPorts „AOut“ und „BOut „ folgende Funktionsaufrufe in den RTE-Dateien vorweisen.

- In: Rte_Read_SWC_In_DataType
- AOut: Rte_Write_SWC_AOut_DataType
- BOut: Rte_Write_SWC_BOut_DataType

4.3.3 Automatisiertes Task Mapping

Bei der Erstellung von Applikationen ist es notwendig, im entsprechenden AUTOSAR-Modellierprogramm die Zuordnungen zwischen den möglichen Tasks und den vorhandenen Runnables herzustellen. In SystemDesk von der Firma dSpace kann diese Zuordnung per Mausklick oder automatisch konfiguriert werden. Normalerweise stellt das Betriebssystem mit Hilfe des Scheduler-Moduls (2.1.4) Tasks zur Verfügung, welche dann die vom RTE-Generator (2.1.3 RTE-Generierung) erzeugten Taskbodies ausführen.

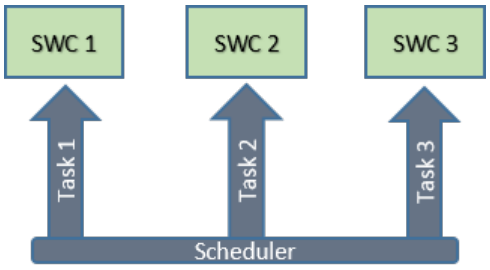

Abbildung 4.5: Task-Mapping

Diese Informationen werden jedoch nicht beim Exportieren einer ARXML berücksichtigt und müssen somit anderweitig beschafft werden. Ohne dieses „Wissen“ ist es später nicht möglich, ein Runnable-Scheduling für die Simulation automatisch zu generieren. Eine Lösung bieten die generierten Taskbodies in der RTE. Wird auf die entsprechende Quellcode-Datei ein Suchalgorithmus angewendet, können die Beziehungen zwischen Runnable und Task abgespeichert werden.

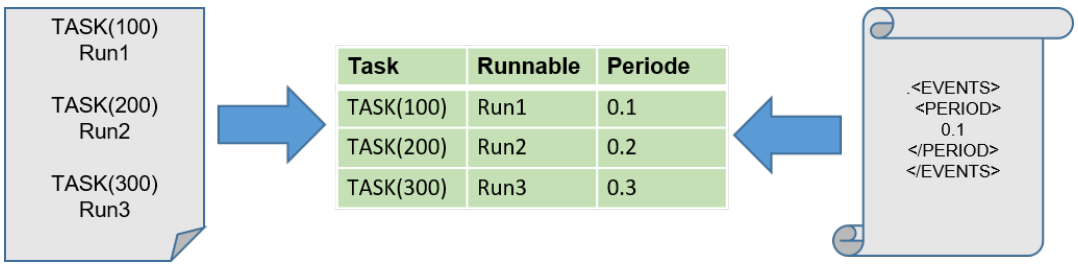

Abbildung 4.6: Taskmapping

Der Algorithmus kann so realisiert werden, dass nach den Runnable-Funktionsnamen gesucht und anschließend geprüft wird, in welchem Taskbody sich diese befinden. Anhand der Beziehungen kann anschließend eine Tabelle erzeugt werden, in der die Task-Runnable-Zuordnungen jederzeit abrufbar sind. Die Tabelle kann später durch

weitere zum Ausführen von Tasks notwendigen Informationen aus der ARXML ergänzt werden. Wie in Abbildung 4.6 zu sehen, kann das z. B. die Periode, also das zeitliche Aufrufen von Runnables, sein.

4.3.4 RTE-Dateien zuweisen

Damit zu testende Daten später in der Simulation an den richtigen Stellen zur Verfügung gestellt werden können, müssen diese zum entsprechenden Zeitpunkt bekannt sein. Soll z. B. ein Testwert geschrieben werden, muss die Stelle an der sich die dazugehörige Funktion befindet, bekannt sein. Diese Funktionen werden durch das korrespondierende Portkommando aufgerufen. Dazu wird den generierten SWC-Objekten (siehe 4.2.1) ein weiteres Attribut zur Verfügung gestellt. Das ist der Verweis zu der RTE-Datei, welche das zugehörige Portkommando enthält. Dadurch wird es dem Nutzer ermöglicht, bei der Konfiguration der Testdaten einen einzelnen Port zu stimulieren oder auf diesem zu lauschen. Die Testkonfiguration wird im Kapitel 4.5 näher erläutert.

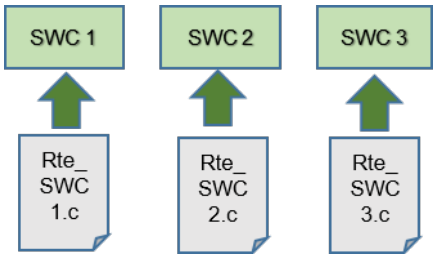

Abbildung 4.7: Rte-Dateien zuweisen

Die Abbildung zeigt wie die Rte-Dateien automatisiert zugewiesen werden können. Der generierte Code von SystemDesk sieht Beispielsweise so aus, dass für die jeweiligen Softwarekomponenten und ihre Schnittstellen eine separate Quellcode-Datei angelegt wird. Anhand des Namen der Datei, kann folglich das SWC-Objekt ermittelt werden. Der bereits erwähnte Verweis im Objekt speichert den gesamten Pfad zur Datei. Die Lokalisierung des Codes ist für spätere Zwecke folglich abgeschlossen.

4.4 Visualisierung von Informationen

Die Präsentation aller relevanten Informationen über das zugrundeliegende Testobjekt soll dem Testenden eine schnelle Übersicht des Gesamtsystems bieten. Die Usability ist folglich auch ein wichtiger Punkt im Konzept, da es so ermöglicht werden soll, auch einem Benutzer ohne spezielle Kenntnisse zu AUTOSAR oder der Programmierung einen Test konfigurieren und ausführen zu lassen. Relevante Informationen sind alle VFB-Komponenten (siehe 2.1.2), welche aus der XML gewonnen werden und deren Überprüfung auf spezifische Vorgaben.

4.4.1 Visualisierung der Komponenten

Der Visualisierer erleichtert die Übersicht über das Projekt, in dem alle relevanten Komponenten grafisch dargestellt werden. Softwarekomponenten und ihre Bestandteile sind somit direkt ersichtlich. Weiterhin werden die Verbindungen unter den SW-Cs dargestellt, was wieder dem VFB entspricht.

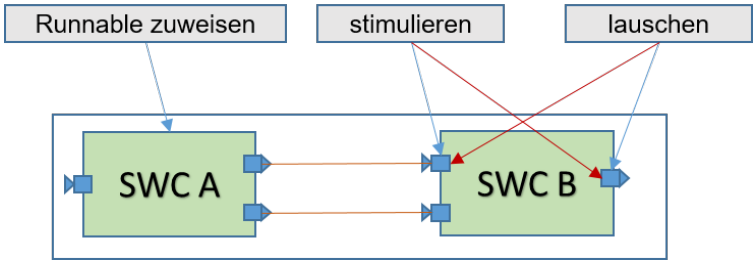

Abbildung 4.8: Visualisierung und Konfiguration

Abbildung 4.8 zeigt die Optionen, die ein Benutzer verwenden kann. Diese werden per Rechtsklick mit der Maus aufgerufen.

4.4.2 Anzeigen von Systeminformationen

Weiterhin soll dem Benutzer über eine Konsole eine komplette Systemdiagnose bereitgestellt werden. Die ersten Hinweise, wie z. B. der RTE- und Compiler-Pfad, werden bereits beim Start angezeigt. Beim Laden einer XML wird die Information über die Validierung des Dokuments ausgegeben (siehe 4.1). Weiterhin werden alle, wie im Kapitel 4.2.1 erläutert, Überprüfungen angezeigt. Hierbei dürfen auch nicht die in Kapitel 5.3 Analyse der RTE-Quellcode-Dateien fehlen. Die Konsole dient außerdem der Anzeige der gewünschten Resultate während der Simulation.

4.5 Konfigurierung der Testumgebung

Für eine benutzerfreundliche Konfigurierung der Testumgebung wird eine Oberfläche bereitgestellt, welche das einfache Stimulieren von Ports und Erzeugen von Testwerten bietet. Der Nutzer kann vorerst zwischen eigens festgelegten Testdaten oder einer Grenzwertanalyse wählen. Zusätzlich sollen alle Tasks in der RTE angezeigt und ausgewählt werden können. Dadurch wird die Möglichkeit geboten, gezielt Basissoftwarefunktionen anzusprechen und deren Rückgabe während der Simulation ausgeben zu lassen. Das Stimulieren wird, wie in Abbildung 4.8 angedeutet, über einen Rechtsklick auf den entsprechenden Port ermöglicht. Dabei öffnet sich ein Kontextmenü, welches die Optionen für das Stimulieren und das Lauschen anbietet. Ein weiteres Kontextmenü, welches über einen Rechtsklick auf die Softwarekomponente geöffnet wird, bietet das Zuweisen der erforderlichen Runnable- und RTE-Dateien.

4.6 Generierung der Testumgebung

Die Generierung der Testumgebung erfordert nicht nur die Informationen, die aus dem Import gewonnen werden, sondern auch die Konfiguration des Gesamtsystems und das Hinzufügen benötigter Ressourcen, wie Testdaten oder fehlende Header-Dateien. Für die erfolgreiche Generierung der Testumgebung reichen die Informationen aus der Spezifikation allein nicht aus. Vielmehr wird ein Beispiel benötigt, welches die genaue Implementierung der generierten RTE-Dateien aufzeigt. Da am Lehrstuhl die Möglichkeit besteht, Applikationen selbst zu entwerfen und daraus eine ARXML und RTE zu generieren, wird eine vorerst einfache Anwendung erzeugt. Dies wird dann entsprechend untersucht, um einen generischen Ansatz zur Isolierung des Testobjektes zu finden. Im Folgenden werden die vier notwendigen Schritte

1. Verständnis der Abhängigkeiten zur Basissoftware
2. Entwicklung einer Beispielapplikation
3. Analyse der generierten RTE
4. Isolierung des Testobjektes

erläutert, um das Testobjekt in einer gesonderten Umgebung auszuführen.

Die generelle Ausführung von Applikationen auf der ECU erfordert das Vorhandensein aller Quellcodedateien. Das betrifft die Runnables, die RTE und die Basissoftware. Die Kapselung der Applikation und der RTE vom Rest des AUTOSAR-Schichtenmodells hat zur Folge, dass nur Applikations- und RTE-Dateien compiliert werden können.

Die zu testenden Daten werden im Konfigurations-Dialog vom Benutzer festgelegt (4.5 Konfigurierung der Testumgebung).

Abhängigkeiten zur BSW

Die Verbindungen zu den Basissoftware-Modulen betrifft die Schichten RTE und Basissoftware im AUTOSAR-Standard-Modell. Das liegt daran, dass die Softwarekomponenten in der Applikationsschicht eine Kommunikation über die RTE als standardisierte Schnittstelle verwenden müssen ([AUT14h]). Würde eine SWC beispielsweise direkt eine hardwarenahe Funktion aufrufen (z. B. DioWrite), wäre das eine Verletzung der AUTOSAR-Spezifikation. Ein direkter Zugriff ist nur den ComplexDevice-Treibern (4.9) vorbehalten, um eine Echtzeit-Interaktion mit der Hardware zu garantieren.

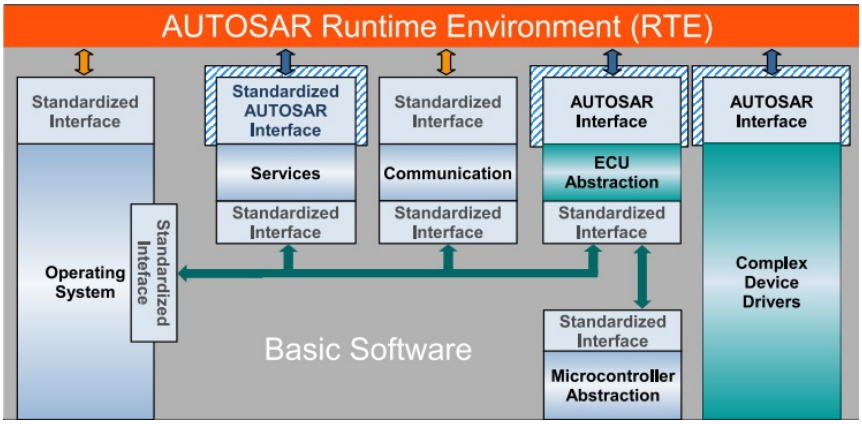

Abbildung 4.9: Verbindungen zur BSW vergl. [AUT14h]

Abbildung 4.9 verdeutlicht noch einmal die Abhängigkeiten zu den Basis-Software-Modulen. Eine wertvolle Voraussetzung für diese Arbeit sind die standardisierten Interfaces (**Standardized Interface**) und werden daher folgend untersucht. ECU-Abstraction und ComplexDeviveDrivers werden in dieser Arbeit nicht berücksichtigt. „**Communication**“ wurde bereits in 2.1.4 beschrieben.

OS und Services

Der Spezfikation [AUT14e] für das **Operating System** ist zu entnehmen, dass eine Kommunikation über die bereits erwähnten Taskbodys erfolgt. Über Services wie „ActivateTask()“ kann dem OS mitgeteilt werden, dass ein Weiterer Task auszuführen ist (siehe [AUT14h]). Die **Services** bieten mit standardisierten Methodenaufrufen die Möglichkeit aus der Applikationsschicht mit Basissoftwaremodulen zu interagieren ([KF09]). Ein Service wurde bereits in 2.1.4 ECU-Statemanager erläutert.

Entwicklung anhand einer Beispiel-Applikation

Da das Konzept einen theoretischen Weg zur Realisierung einer Testumgebung bereitstellt, soll dies auch praktisch analysiert werden. Hierfür wird eine Applikation benötigt, welche so einfach wie möglich gehalten werden soll, um einen ersten Weg zu einer isolierten Applikation zu ebnen. Mit SystemDesk (dSpace) wird ein einfacher Addierer modelliert und die zugehörige RTE und Basissoftware generiert. Anschließend wird die korrespondierende ARXML mit sämtlichen Systeminformationen exportiert. Somit wird der gesamte Input (RTE + XML) der zu testenden Applikation für den Simulator bereitgestellt. Wie in Kapitel 3.2 erläutert, ermöglicht SystemDesk in Kombination mit VEOS das Testen des generierten Steuergeräte-Codes. Die Ergebnisse der Simulation mit SystemDesk werden später mit den Ergebnissen der Eigenimplementation verglichen.

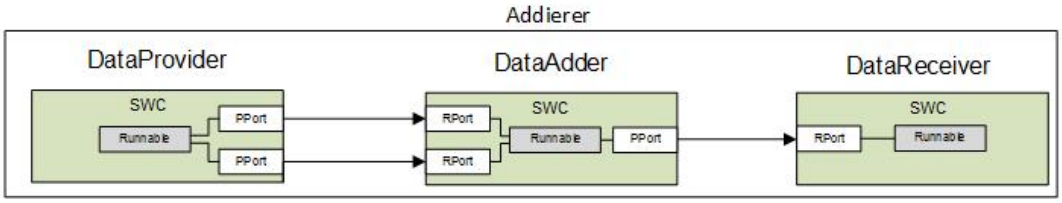

Abbildung 4.10: Applikation Addierer

Die obere Abbildung zeigt die ausführbare Applikation Adder die später ohne Abhängigkeiten zur Basis-Software ausgeführt werden soll. Mit einer Testumgebung, welche die Aufgaben vom eigentlichen OS und so das Scheduling der Tasks übernimmt, soll eine Simulation realisiert werden. Weiterhin übergibt die Testumgebung die entsprechenden Testdaten, die vor dem Simulieren vom Benutzer festgelegt werden können. Die Applikation realisiert einen einfachen Addierer, der aus drei Softwarekomponenten besteht. Die Runnable in der Softwarekomponente **DataProvider** erzeugt Werte und übergibt diese mit Hilfe der RTE an die SWC Adder. Die Runnable von Adder addiert die beiden übergebenen Werte miteinander und übergibt das Ergebnis an die SWC DataReceiver. Auch dies geschieht wieder mit Hilfe der RTE. Die Anwendung kann später durch die bereits erwähnte Testumgebung beobachtet und stimuliert werden. Wichtig hierbei ist ein Unit-Test, der DataAdder gesondert mit Testwerten speist und und diese am Ergebnissport ausließt

Analyse der SystemDesk-RTE

Nach der Generierung der RTE mit SystemDesk liegen alle benötigten Dateien im entsprechenden Verzeichnis vor.

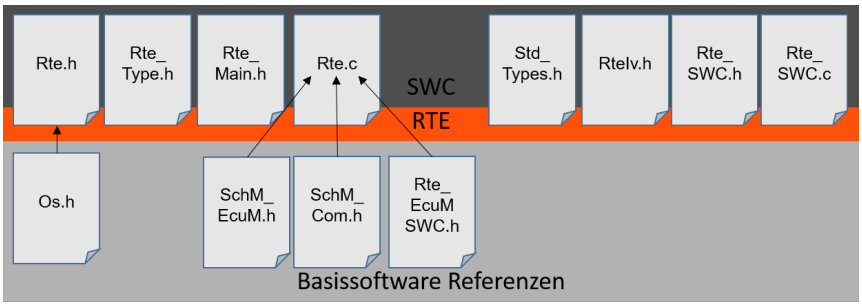

Abbildung 4.11: Generierte RTE-Dateien

Die Abbildung zeigt die generierten RTE-Dateien und die Abhängigkeiten zur Basissoftware auf Quellcode-Ebene. Die Farben orientieren sich dabei am AUTOSAR-Schichtenmodell ([AUT14h]), um so ein besseres Verständnis der Abhängigkeiten zu illustrieren. Die Dateien **RteIv.h**, **Rte_SWC.h**, **Rte_SWC.c** bringen die Funktionalität der Softwarekomponenten und das verwalten von Inter-Runnable-Variablen mit. **Rte_Main.h**, **Rte_Type.h** und **Std_Types.h** verfügen über die RTE-Datentypen und die Start-/Stopfunktion der Run-Time-Environment.

Rte.h Diese Header-Datei dient der Rte.c als Funktions- und Variablen-Deklaration. Im Bild 4.11 ist die Abhängigkeit zum Basis-Software-Betriebssystem zu sehen. Dies wird durch das Einbinden (*include<Os.h>*) von Os.h ersichtlich.

Rte.c Laut AUTOSAR-Spezifikation [AUT14h] ist das C Modul welches die Generierte RTE beinhaltet, mit „Rte.c“ zu benennen. Das bedeutet, hier werden die eigentlichen Standard-Interfaces zur Kommunikation mit anderen Modulen implementiert. Die Betrachtung der SystemDesk-RTE bringt eine etwas abweichende aber wesentlich übersichtlichere Methode zur Spezifikation zum Vorschein. So werden die Standard-Interfaces der jeweiligen Softwarekomponenten als extra C-Datei generiert (**Rte_SWC.c**). Zur besseren Veranschaulichung soll das Beispiel Adder aus dem Grundlagenkapitel 2.1.2 dienen. Abbildung 4.11 zeigt außerdem, wie die Services mit dem RTE-Modul verbunden sind. Die Abhängigkeiten zu den Services werden durch **SchM_EcuM.h**, **SchM_Com.h** und **Rte_EcuMSWC.h** implementiert.

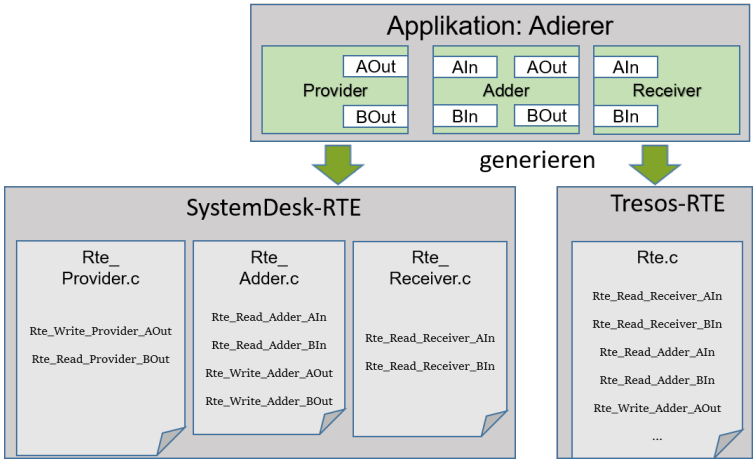

Abbildung 4.12: genrierte SWC-Dateien

Die Grafik 4.12 zeigt die einzelnen Softwarekomponenten der Applikation Adder, welche hier auch als Referenz-Applikation dient. Die Schnittstellen der einzelnen SWCs werden separat in Quellcode-Dateien implementiert. Dabei ergibt sich folgende Namenskonvention, wie in der Abbildung ersichtlich.

`Rte_SWC-NAME.c`

Im Kapitel 4.3.4 RTE-Dateien zuweisen wurde beschrieben, wie jedem SWC-Objekt eine RTE-Datei zugewiesen wird. Da jetzt die Namenskonvention beim Generieren der RTE mit SystemDesk bekannt ist, kann dies für ein generisches Zuordnen der Dateien zum SWC-Objekt ausgenutzt werden. Für die Applikation werden also noch die Dateien Rte_Provider.c, Rte_Adder.c und Rte_Receiver.c generiert. Ein Vergleich mit einer generierten RTE von **Tresos**, einem Tool zum Generieren von Basissoftware und RTE, hebt diesen Unterschied hervor. Damit werden alle Interfaces nach Spezifikationsvorgaben in der Rte.c Datei implementiert (Abbildung 4.12). Weiterhin befinden sich hier die generierten Taskbodys. Diese ermöglichen überhaupt erst den Start und die Ausführung der Applikation, denn in den Taskbodys befinden sich die Runnables (2.1.2). Davor müssen aber noch alle Variablen (Signale) initialisiert werden. Dies geschieht mit der Methode `Rte-Start`, welche folglich zuallererst ausgeführt werden muss. Abschließend ist noch die Funktion `Rte-Stop` zu erwähnen, welche die De-Initialisierung der Signale realisiert.

4.6.1 Isolierung des Testobjektes

Wie im Grundlagenkapitel 2.2.3 erläutert, bedarf es eines Testrahmens, um den Prüfling von seinen normalen Abhängigkeiten zu trennen und in einer gesonderten Simula-

tionsumgebung zu validieren. Es müssen aber erst einmal alle Abhängigkeiten bekannt sein. Dazu ist es zum einen hilfreich die Vorgaben der AUTOSAR-Spezifikationen zu beachten und zum anderen nicht spezifizierte Verbindungen herauszufinden.

Im vorhergehenden Abschnitt wurde bereits dargelegt, welche RTE-Dateien für die Applikation benötigt werden. Bei der RTE-Generierung werden aber auch Header-Dateien für die Bekanntmachung der Funktionen der entsprechenden BSW-Module erzeugt. Weil letztere nicht zur Verfügung stehen, würden die angesprochenen Header beim Compilieren viele Fehler verursachen. Das liegt unter anderem an den Verweisen zu den Funktionen oder Variablen aus der BSW. Die Header-Dateien werden wie in C/C++ üblich, fest inkludiert. Damit das Testobjekt nicht verändert wird (z. B. auskommentieren der Includes) werden hierfür Dummy- und Stub-Header (2.2.3) benötigt.

Generieren von Dummys und Stubs

Dummys werden für die Header, welche nicht benötigte Funktions-Deklarationen aufweisen, gebraucht. Dazu gehört z. B. *Rte_EcuMSwc.h* und alle Verweise, die nicht den Standard-Dateien entsprechen. Die benötigten Basissoftware-Header-Dateien deklarieren zwar die benötigten Funktionen, verweisen aber ebenfalls wieder auf weitere Header, die nichts mit der Applikation zu tun haben.

Stubs interagieren daher als Header mit benötigten Funktionen, bspw. um den Nutzer darüber zu informieren, dass eine Service-Funktion aufgerufen wurde. Die implementierte Funktion kann in einer selbst erzeugten RTE-Datei mit untergebracht werden. Durch das Vorhandensein dieser Dateien meldet der Pre-Prozessor des Compilers keinen Fehler. Hierbei ergibt sich allerdings ein neues Problem. Der Nutzer müsste händisch alle nicht benötigten Dateien entfernen, da durch die Generierung der Dummy und Stubs die Dateien doppelt vorhanden wären. Z. B. würde die Datei `SchM_Com.h` einmal Original und einmal generiert vorhanden sein. Eine mögliche Lösung wäre, ein weiteres Verzeichnis anzulegen, in das alle zu verwendenden originalen Dateien kopiert werden und anschließend die Dummys und Stubs hinzukommen. Das würde gleichzeitig den Vorteil bieten, dass die ursprünglichen Dateien nicht verändert werden können.

Benachrichtigung über den Aufruf von BSW-Funktionen

Während der Simulation kann es für den Testenden hilfreich sein, wenn er darüber informiert wird, ob und wann die Applikation Basissoftware-Funktionen aufruft. Dazu werden weitere Stubs generiert, welche funktional den Basissoftware-Modulen entsprechen. Für den normalen Ablauf sind das die Services vom ECU-Statemanager und

dem COM-Modul. Im Vorhergehenden Unterkapitel wurde eine RTE genauer analysiert. Dabei stellte sich heraus, dass die Initialfunktionen Com_MainFunctionRx und Com_MainFunctionTx aus dem COM-Modul ([AUT14b]) beim Start aufgerufen werden. Auch die Funktionen Com_SendSignal und Com_ReceiveSignal ([AUT14b])werden eventuell benötigt. Diese Methoden werden dem COM-Dummy mitgegeben und mit dem Aufruf dieser Funktionen soll dem Nutzer eine entsprechende Meldung angezeigt werden.

4.6.2 Bereitstellen und Auslesen der Testdaten

Wie in Kapitel 4.3.4 RTE-Dateien zuweisen bereits erwähnt, ist die Information über den Standort der aufzurufenden Funktionen von hoher Relevanz, denn hier werden die Testdaten direkt übergeben.

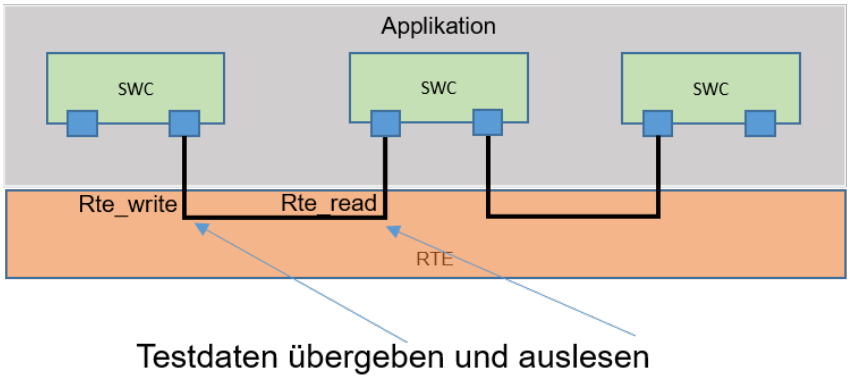

Abbildung 4.13: Übergeben und Auslesen der Testdaten

Die Abbildung verdeutlicht den Aufruf der Schreib- und Lesefunktionen. Mit Hilfe des entsprechenden Portkommandos wird die zugehörige Funktion aus der RTE aufgerufen (**Rte-read**). In dieser Methode kann durch das Manipulieren der Übergabeparameter das Einbringen von Testdaten realisiert werden. Stimuliert der Nutzer einen Port, wird die entsprechende Datei und das zum Port zugehörige Portkommando (Funktion) ermittelt. Der Funktion werden dadurch die Testdaten übergeben. Gleiches gilt auch für das Lauschen an einem Port, wobei hier die Ergebnisse eines Aufrufs (return Wert) ausgelesen werden.

Testdatenvektor hinzufügen

Die vom Nutzer generierten Testwerte müssen der Simulationsumgebung bekannt sein. Dazu nimmt ein Vektor die entsprechenden Daten auf. Somit ist es dem Ar-

XmlAnalyzer möglich, eine neue Ressourcen-Datei zu erzeugen, welche später mit kompiliert wird.

4.6.3 Umsetzung Task-Verwaltungsmodul

Wie oben beschrieben, ist das RTE-Modul (Rte.c) zentrales Glied für die Ausführung der Applikation. Daher gilt es, die Funktionen und Taskbodys in diesem auszuführen. Dies wird mit einem provisorischen Task-Verwaltungs-Modul realisiert. Das Scheduler-Modul in der Basissoftware wird vom AUTOSAR-Betriebssystem (z. B. OSEK) benötigt, um das sequentielle und parallele Ausführen von Runnables und BSW-Modulen zu realisieren 2.1.4. Da dieses Modul Bestandteil der Basissoftware ist, steht ein Scheduling der Tasks nicht mehr zur Verfügung. Daher wird eine eigens implementierte Task-Verwaltung benötigt, welche das Ansprechen der Taskbodys und somit die Ausführung der Runnables ermöglicht. Die Realisierung erfordert das Verständnis des Verwaltens und Ausführens von Tasks in AUTOSAR (siehe 2.1.4). Der RTE-Generator generiert in der C-Datei *Rte.c* so genannte Task-Bodys, welche die Aufrufe der Runnables beinhalten (2.1.3). Diese Generierung geschieht AUTOSAR-spezifisch, weswegen jeder Code-Generator die gleiche Implementierung berücksichtigen muss. Diese „Vorschrift“ wird im selbst zu implementierenden Task-Verwaltungsmodul ausgenutzt, um so ein generisches Ausführen der Taskbodys zu realisieren. Außerdem werden hier die RTE-Start- und RTE-Stop-Funktion aufgerufen. Im Kapitel 5.3 wurde beschrieben, wie die Informationen über das Taskmapping beschafft werden. Dabei kann an dieser Stelle auch die Analyse des Zeitverhalten realisiert werden, in dem mit dem Starten der RTE auch ein Zeitzähler mitläuft, welcher zu jedem Aufruf von Funktionen festgehalten werden kann. So soll jedes Ereignis (Schreiben und Lesen) mit einer Auflösung im Millisekunden-Bereich protokolliert werden können.

Abbildung 4.14: Ergebnis- und Datenvektor

Wie in Abbildung 4.14 zu sehen, wird pro Port ein Datenvektor bereit gestellt. Für einen PPort ist zusätzlich ein Zeitvektor vorhanden, in dem bei jedem Ergebnis die Zeit festgehalten wird. Hierbei ist das Interessante wie sich parallel aufrufende Runnables an einem festen Zeitpunkt verhalten.

4.7 Entwurf des ArXmlAnalyzer

Die Verarbeitung der Informationen aus einer XML-Datei und das Generieren einer Testumgebung erfordert ein Programm, welches die Dateien einließt, auf Spezifikationsvorgaben prüft und die gewonnenen Daten für weitere Zwecke bereitstellt. Der ArXmlAnalyzer hat die Aufgabe, die entsprechenden Informationen in der XML-Datei zu finden und diese zu verwalten. Aus den Applikationsinformationen werden Objekte generiert, die anschließend auf Spezifikationsvorgaben geprüft und miteinander verbunden werden. Weiterhin sollen alle Softwarekomponenten und deren Verbindungen direkt visualisiert werden. Wichtigster Bestandteil ist die Erkennung der RTE-Dateien und die Erweiterung letzterer.

Die Anforderungen hierbei sind die Erstellung der Systemarchitektur nach **Model-View-Controller** (MVC) Prinzip. Es wird demzufolge ein Datenmodell (model) benötigt, das alle projektrelevanten Informationen bereitstellt, wie einen Controller, der für die Logik und die Kommunikation der einzelnen Bestandteile verantwortlich ist und die Benutzerschnittstelle (View), die es ermöglicht, die Interaktion zwischen Programm und Benutzer zu realisieren. Nach dem MVC-Muster ist die View „dumm“ und stellt lediglich die Benutzersteuerelemente bereit, welche die entsprechenden Daten aus dem Modell darstellt.

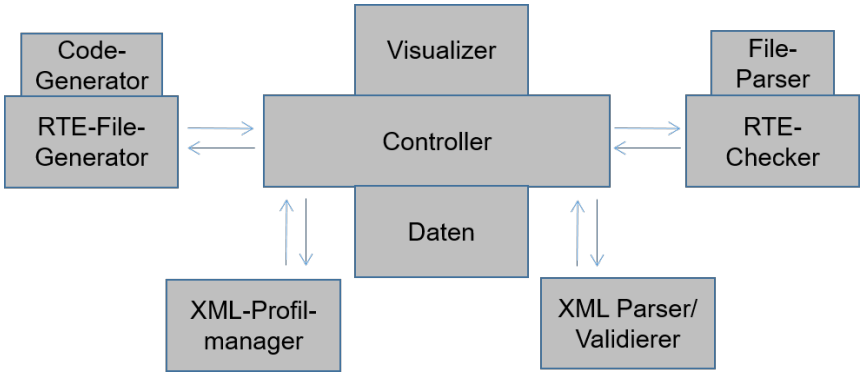

Abbildung 4.15: Programmentwurf

Die obere Abbildung zeigt den einfachen Entwurf des Programms. Hierbei ist zu erkennen, dass neben dem Datenmodell, welches alle Projektinformationen persistent hinterlegt, auch weitere Komponenten mit dem Controller interagieren. Der Controller ist demzufolge die oberste Instanz und regelt alle Interaktionen der Module.

Der Parser hat die Aufgabe, alle relevanten Informationen aus der ARXML-Datei zu lesen und daraus Objekte zu generieren (siehe 5.2). Die gewonnenen Informationen werden durch den Contoller im Datenmodell abgelegt. Dort stehen sie ab da an jeder Zeit für weitere Operationen bereit.

Der **XML-Profilmanager** deckt die Aufgaben, welche im Kapitel Datenimport beschrieben wurden, ab. Die Anforderungen für die Validierung (4.1), werden durch den **Validierer** erfüllt. Der **RTE-Checker** bringt die Funktionalität für das Sammeln von Informationen aus den RTE-Quellcode-Dateien mit, wie es in dem Abschnitt 5.3 Datenanalyse beschrieben wurde. Mit dem **RTE-File-Generator** und dem **Code-Generator** bekommt das Programm ein Werkzeug, welches das Generieren aller benötigten Dateien und das Übergeben der Testdaten an die entsprechenden Funktionen ermöglicht (5.6). Das Parser- und RTE-Checker-Modul werden direkt als Klassenbibliothek bereitgestellt. Somit kann jeder Entwickler auf diese Bibliotheken zurückgreifen oder erweitern.

4.8 Einschränkungen

Aufgrund des Zeitrahmens und der Forschung an vorerst kleinen Applikationen steckt dieses Projekt noch in den Kinderschuhen. Daher verdeutlicht dieses Unterkapitel die Einschränkungen, die sich bei diesem Konzept ergeben.

4.8.1 Zeitverhalten

Die Simulation einer Applikation auf dem Windows-PC kann zeitlich nicht mit dem Ablauf auf der Zielhardware verglichen werden. Folglich kann die Analyse der Echtzeit nicht als Kriterium für diese Arbeit betrachtet werden.

4.8.2 Beschränkung auf SystemDesk RTE

Die SystemDesk-RTE hat sich im Gegensatz zu anderen als gut verständliche Referenz herausgestellt. Dies spiegelt sich vor allem in der automatisierten Kompilierung aller Dateien wieder. Allerdings bringt das auch den großen Nachteil mit, dass dieser Simulator nur auf mit SystemDesk generierte RTE's beschränkt ist. Weiterhin kann nicht sichergestellt werden, dass Projekte, die größer als die getesteten Applikationen sind, auch kompilierbar sind. Da die Applikation Adder mit Timing- und DataReceived-Events realisiert wurde, werden vorerst auch nur diese berücksichtigt.

4.8.3 Veränderung des Testobjektes

Im Kapitel 4.6.2 Bereitstellen und Auslesen der Testdaten wurde beschrieben, wie die Testdaten für die Stimulation der jeweiligen Ports bereitgestellt werden. An dieser Stelle werden dem Testobjekt „Fremd-Informationen“ hinzugefügt. Es werden demzufolge neue Code-Zeilen in die betroffenen RTE-Dateien geschrieben.

4.8.4 Taskverwaltungsmodul

Dieses Modul wurde konzeptioniert, um die RTE-Task-Bodys, wie in der AUTOSAR-Spezifikation [AUT14h] beschrieben, anzusprechen und somit die Applikation auszuführen. Das Verhalten kann nicht mit einem richtigen Scheduler wie in der Basissoftware verglichen werden, da dies den Rahmen dieser Arbeit sprengen würde.

4.9 Zusammenfassung

Im Konzept wurde eine möglich Lösung für das Testen von AUTOSAR-Applikation vorgestellt. Mit Hilfe einzelner Phasen, die für die Realisierung dieser Arbeit ausschlaggebend sind, sollte dem Lesenden ein Bezug zu den jeweiligen Anforderungen und deren Lösung gegeben werden. Das Validieren der Informationen wurde hierbei in XML-typischen Überprüfungen überführt. Das Importieren der Projektdaten wird durch einen geeigneten XML-Parser realisiert, welcher durch einen Profilmanager eine Möglichkeit bietet, auch andere AUTOSAR-Xml-Versionen zu lesen. Die Analyse der RTE betrifft sowohl das Vergleichen der ARXML- und der RTE-Dateien, als auch das automatisierte Erkennen der einzelnen Taskmappings und das Zuweisen der entsprechenden RTE-Dateien zu ihren SWCs. Die Visualisierung der geparsten Informationen bietet dem Anwender eine schnelle Übersicht und erlaubt das Stimulieren und Abhören beliebiger Ports. Dadurch können auf einfache Weise System- und Unit-Tests definiert werden. Auch das Erzeugen beliebiger Testwerte wird unterstützt. Die letzte Phase (Generierung der Testumgebung) hat aufgezeigt, wie die Abhängigkeiten zur Basissoftware simuliert werden können. Das Erstellen eines Prototypen mit SystemDesk und die darauffolgende Analyse der daraus generierten RTE haben die Voraussetzungen geschaffen, das Testobjekt vom Rest des Steuergeräte-Codes mit Hilfe von Dummy- und Stub-Dateien zu isolieren. Anschließend wurde ein Weg beschrieben, wie die Testdaten an die gewünschten Ports übergeben werden können. Die Ausführung des Testobjektes wird durch das Task-Verwaltungsmodul realisiert, welches das parallele Ausführen von Funktionen unterstützt. Dem Lesenden wurde dabei eine Technik erläutert, die es ermöglicht, gezielt Taskbodys im RTE-Modul anzusprechen. Zum Schluss wurde die Umsetzung des Programms vorgestellt, das alle Aufgaben der einzelnen Kapitel realisieren soll.

5 Implementierung

Im Folgenden wird die Umsetzung des ArXmlAnalyzer beschrieben. Die komplette Implementierung wird mit C# und dem *.Net Framework* realisiert. Während die Applikation selbst in C/C++ generiert wird. Ähnlich wie im Konzept wird sich für die Gliederung der Unterkapitel wieder auf die einzelnen Phasen im Projekt bezogen. Somit soll es dem Leser ermöglicht werden, die Implementierung der gefundenen Lösungen aus dem Konzept nachvollziehen zu können.

5.1 XML-Validierung

Die im Konzept beschriebene Überprüfung auf eine wohlgeformte und AUTOSAR-Konforme XML Datei, wird mit dem Validater realisiert. Dieser befindet sich in dem Package Validation, in welchem auch der ValidateDialog für die Nutzerinteraktion bereit steht. Über den Dialog kann der Pfad zur ARXML-Datei und der korrespondierenden Schema-Datei festgelegt werden. Anhand dieser Informationen kann der Validater anschließend eine Schemaprüfung (2.1.1) durchführen. Daraus folgt dass, eine Validierung jederzeit möglich ist.

```
public void validate(string xml, string xsd)
{
  XmlSchemaSet schemas = new XmlSchemaSet();
  schemas.Add("", xsd);
  ...
  XDocument doc = XDocument.Load(xml);
  bool errors = false;
  doc.Validate(schemas, (o, e) =>
  {
    OnNewConsoleMessageReceived(e.Message);
    errors = true;
  });
}
```

Listing 5.1: Ausschnitt aus der Funktion validate

Mit Hilfe des EventHandlers NewConsoleMessageReceived (5.4.3) können alle Meldungen an die Konsole geschickt werden. Das Herzstück ist hierbei die Methode **validate**. Dieser werden zum einem die ARXML-Datei und die Schema-Datei übergeben. In der Funktion wird eine Instanz vom Typ **XmlSchemaSet** ([Küh13]) erzeugt und der Pfad zur Schemadatei übergeben. Mit Hilfe einer Instanz von **XDocument** wird die ARXML gelesen und durch die Funktion validate mit dem Schema verglichen. Das Ergebnis wird dem Akteur anschließend auf der Konsole präsentiert. Ist die Validierung erfolgreich können die Daten importiert werden.

5.2 Datenimport

Der Import von Informationen für eine lauffähige Testumgebung spaltet sich in zwei Themengebiete. Zum Einen werden durch das Parsen der ARXML-Datei alle relevanten Informationen zum korrekten Zusammenbau eines Zielsystems bereitgestellt. Zum Anderen müssen ebenfalls Informationen aus entsprechenden RTE-Quellcode-Dateien berücksichtigt und erweitert werden. Die folgenden Unterabschnitte beschäftigen sich daher mit der Implementierung zur Gewinnung der erwähnten System-Informationen.

5.2.1 Generieren von Objekten

Die relevanten Informationen aus der ARXML müssen jederzeit zugänglich sein. Gerade benötigte Daten sollen nicht jedes mal neu geparst werden, weil so ein größerer Rechen- und Speicheraufwand entstehen würde. Deshalb werden die Informationen einmal komplett ausgelesen und in Objekten gespeichert. Hierfür gibt es vordefinierte abstrakte Datentypen, die im Projekt bereit liegen. Von diesen wird beim parsen direkt ein Objekt mit allen gefundenen Informationen erzeugt. Die genaue Funktionsweise der Objektgenerierung während der Informationsgewinnung, ist Bestandteil des nächsten Abschnitts 5.2.2.

5.2.2 XML-Parser

Der ArXml Parser wurde als Klassenbibliothek implementiert. Also unabhängig vom Hautprojekt selbst. Die Anforderung, dass auch zukünftige AUTOSAR Versionen einer XML Datei gelesen werden, erfordert zwei Herangehensweisen des Parsers. Zum einen können Knoten in späteren Versionen andere Namen für das gleiche Objekt aufweisen und zum anderen kann sich die Tiefe eines Knotens ändern.

Das **.Net Framework** bietet für das Parsen von XML Dateien unter anderem eine Technologie Namens **Linq** (”Language-Integrated Query“) to XML, siehe[Küh13]. Dies ermöglicht das Abfragen eines XML Dokuments mit SQL-Syntax. Dazu wird die XML-Datei als so genanntes ”XDocument“in den Arbeitsspeicher geladen. Anschließend können entsprechende Abfragen getätigt werden. Z.b.

```
from appswc in xmlDoc.Descendants(tagName)
select new SWC
{
  Shortname = getNode(appswc,Profile.TagDict[”ShortName”]).Value,
  ...
}
```

Listing 5.2: Abfrage eines Knotens

Hierbei werden aus dem XML Dokument `xmlDoc` alle Nachfolger Knoten des angegebenen TagName (`tagName`)ausgelesen und in Variablen geschrieben. Mit der

Anweisung „`new SWC`“ wird direkt ein Objekt der Klasse `SWC` erstellt. Außerdem ist im Code Beispiel zusehen, dass die `getNode` Methode den Wert des gesuchten Tags in die Variable `Shortname` schreibt.

Der ARXML-Parser wird als Klassenbibliothek bereitgestellt. Das erspart einem anderen Entwickler welcher ebenfalls ARXML-Dateien auslesen muss, eine Eigenimplementierung. Diese Bibliothek kann im Visual Studio einem Projekt als Verweis zugeordnet werden und direkt verwendet werden. Die Klassenbibliothek bietet aber nicht nur den Parser selber, sondern auch die benötigten Datenstrukturen für die AUTOSAR-Komponenten. Dies setzt die Entwicklung mit C# und dem .netFramework 4.6 voraus.

Versionsunabhängiges parsen

Die Tiefensuche nach bestimmten Knoten wird mit Hilfe der Funktion *getNode* realisiert.

```
public XElement getNode(XElement root, string name)
{
  XElement node = (from xml2 in root.Descendants()
    where xml2.Name.LocalName == name
    select  xml2).FirstOrDefault();
  return node;
}
```

Listing 5.3: SWC getNode

Eine Suche nach einem bestimmten Knoten in der ARXML-Datei wird ebenfalls mit **Linq** umgesetzt. Das heißt die Abfrageumsetzung orientiert sich wieder an einer SQL ähnlichen Syntax.

Im Konzept wurde bereits aufgezeigt, dass die *Tag*-Namen in anderen AUTOSAR Versionen einer generierten ARXML-Datei, unterschiedlich benannt werden. Beim Lesen einer XML wird nach festgelegten Zeichenketten gesucht, um so die Werte oder Attribute zu finden. Hierzu bietet der Parser die Eigenschaft Profile. Diese können zur Laufzeit neu zu gewiesen werden um so ein XML mit andere AUTOSAR-Version parsen zu können. Wird kein Profil zugewiesen, wird nach AUTOSAR-Version 3.2 geparst (Standardprofil). Der Parser bietet für jedes zu parsende AUTOSAR-Element (z. B. SWC) eine eigene Methode also auch die Möglichkeit die Anzahl der Elemente zu beschränken. Weiterhin findet hier auch das Einlesen der Profil-Xml statt.

5.2.3 Profilmanager

Der Profilmanager ermöglicht das Anpassen der einzelnen `Tags` in einer ARXML-Datei. Ein Profil besteht aus einem Profilnamen und einem Dictionary [1], welches alle zu parsenden Tags präsentiert. Ist ein Profil importiert und geladen, kann es zur Laufzeit ausgewählt werden. Anhand dieser Informationen können alle relevanten Knoten im XML Dokument erkannt und ausgelesen werden.

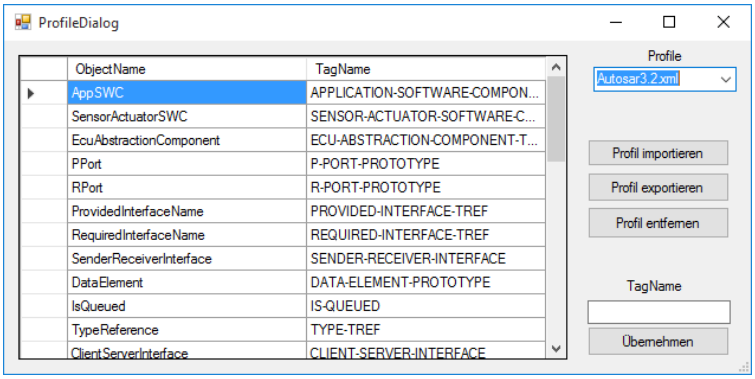

Abbildung 5.1: Test

Abbildung 6.3 zeigt den Profildialog zur Bearbeitung der Tag-Namen. Die Datenstruktur, welche die Datensätze der Tabelle bereitstellt, wird durch das Dictionary im ausgewählten Profil realisiert. Hierbei bildet der **ObjectName** (Tabelle Abb. 6.3)den Key und der **TagName** den korrespondierenden Value. Die Begründung für diese Datenstruktur wird später im Kapitel des Parser-Moduls erläutert. Des weiteren zeigt die Abbildung den Knopf **Profil exportieren**. Bei dem Exportieren eines Profils wird eine entsprechende XML-Datei angelegt.

```
<?xml version="1.0" encoding="utf-8"?>
<TagNmaes>
  <AppSWC>APPLICATION-SOFTWARE-COMPONENT-TYPE</AppSWC>
  <PPort>P-PORT-PROTOTYPE</PPort>
  <RPort>R-PORT-PROTOTYPE</RPort>
  ...
</TagNmaes>
```

Listing 5.4: Ausschnitt aus einer XML Profildatei

Obiges Listing zeigt den Ausschnitt aus einer Profildatei. Diese wird durch eine einfache XML-Datei repräsentiert. Hierbei ist zusehen das die Tiefe der Ebenen nicht größer als eins ist. Somit soll, wie im Konzept bereits erläutert, das Erstellen oder Bearbeiten der Profildatei für jeden Akteur so einfach wie möglich gehalten werden.

[1] Dictionary: Objekte werden per Key Value abgelegt

5.3 Datenanalyse

Die Datenanalyse bezieht sich auf die Quellcode-Dateien der RTE und auf die Objekte, die bei dem parsen der ARXML erstellt wurden. Die folgenden Unterkapitel geben Aufschluss darüber, was genau analysiert werden muss. Eines der wichtigsten Werkzeuge bei der Datenanalyse ist der RTE-Checker.

RTE-Checker

Die RTE spielt im Simulationsprozess eine entscheidende Rolle. Bevor diese aber erweitert, verändert oder neu kompiliert wird, ist es zwingend erforderlich, eine Prüfung der erforderlichen Quellcode-Dateien zu veranlassen. Da die RTE AUTOSAR-spezifischen Merkmalen unterliegt ([AUT14h]), ist es möglich eine generische und automatisierte Prüfung der relevanten Dateien zu realisieren. Zu diesem Zweck wurde der RTE-Checker entwickelt und ebenfalls als Klassenbibliothek zur Verfügung gestellt. Die Bibliothek besteht bis dato nur aus einer Klasse (RTEChecker.cs) und diese implementiert das Interface IWriteToConsole, wodurch die Informationsausgabe an der Konsole (siehe 5.4.3 Systeminformations- und Fehlerausgabe) ermöglicht wird. Die wichtigste Funktion ist getPortCommands, die eine Auflistung aller Portkommandos aus der entsprechenden RTE-Datei aus- und zurück gibt.

```
public List<string> getPortCommands(string file)
{
  ...
  foreach (var item in lines)
  {
    if ((item.Contains("Rte_Write") || item.Contains("Rte_Read")
    item.Contains("Rte_Call")) && item.Contains("Std_ReturnType"))
    {
      list.Add(item);
      OnNewConsoleMessageReceived(item);
    }
    i++;
  }
  OnNewConsoleMessageReceived(i+"␣Portkommandos␣gefunden");
  return list;
}
```

Listing 5.5: Ausschnitt aus der Funktion getPortCommands

Die Funktion bekommt wie im Listing zu sehen, den Pfad zur durchsuchenden Datei übergeben. Diese wird zeilenweise in die Liste **lines** eingelesen. Durch Iteration der Liste kann jede einzelne Zeile (**items**) nach einem Kriterium untersucht werden. Dies wird realisiert in dem Abgefragt wird ob eine Zeile die Sender-Receiver Funktionsaufrufe **Rte_Write** oder **Rte_Read** und den Server-Client-Aufruf **Rte_Call** enthält. Da die zu suchenden Begriffe auch als Kommentar in der Code-Datei stehen kann, wird geprüft ob der Rückgabewert in der selben Zeile vorhanden ist.

5.3.1 ARXML- und RTE-Vergleich

Wie im Konzept angesprochen, ist nicht immer davon auszugehen, dass die RTE-Dateien auch zur entsprechenden ARXML gehören. Daher wird hier die im Konzept

gefundene Lösung zum Vergleich einer XML-Datei mit Quellcode-Dateien implementiert. Die Lösung besteht darin, die Portkommandos der generierten SWC-Objekte (siehe 4.2.1) abzufragen und mit den Funktionsaufrufen in den entsprechenden RTE-Dateien zu vergleichen. Dazu müssen diese Dateien mit einem Parser und einer geeigneten Strategie zum Finden der Portkommandos untersucht werden. Diese werden genau genommen in den zugehörigen Ports gespeichert.

```
public bool isRteValide()
{
  List<string> list = new List<string>();

  foreach (var swc in data.SwcList)
  {
    list.AddRange(rteChecker.getPortCommands(swc.RteFile));
    foreach (var port in swc.PPortList)
    {
      if (!list.Exists(n => n.Contains(port.PortCommand)))
        return false;
    }
    foreach (var port in swc.RPortList)
    {
      if (!list.Exists(n => n.Contains(port.PortCommand)))
        return false;
    }
  }
  return true;
}
```

Listing 5.6: Funktion zum Prüfen einer validen RTE

Das Listing zeigt die Funktion **isRteValid**. Hierbei ist zu sehen dass in einer Liste alle Portkommandos aufgenommen werden, welche der RTE-Checker mit der Funktion getPortCommands gefunden hat. Die genauere Funktionsweise ist dem Kapitel 5.3 RTE-Checker zu entnehmen. In der ersten Schleife werden alle im Projekt befindlichen Softwarekomponenten abgefragt. Da zu diesem Zeitpunkt jede SWC ihre zugewiesene RTE-Datei kennen sollte (Zuweisen von RTE-Dateien), kann diese wie im Listing zu sehen, die zugehörige Datei an den RTE-Checker übergeben und so die Liste der Portkommandos erweitert werden. Das hat gleichzeitig den Effekt, dass die Liste nur so groß wird, wie die Anzahl der Kommandos. Die beiden inneren Schleifen dienen zum Abfragen der jeweiligen Ports (PPort und RPort) einer SWC. Innerhalb dieser wird abgefragt ob das Kommando eines Ports mit einem der gefunden in der Liste übereinstimmt. Hierbei ist zu erwähnen, dass immer geprüft wird ob das Portkommando des Objektes auch in der Datei vorhanden ist und nicht vice versa. Somit soll ausgeschlossen werden, das nach einem Funktionsaufruf gesucht wird, welcher von einem RTE-Generator für andere Zwecke generiert wurde, aber nicht zum eigentlichen System gehört. Anstatt weiterer verschachtelnder Schleifen wird für die eigentliche Abfrage eine Kombination aus Linq und Lambda-Ausdruck verwendet. Linq ermöglicht es hier bei, die Liste mit der Funktion **Exists** und dem übergebenen Kriterium abzufragen. Dabei wird ein boolscher wert zurück gegeben, welcher über das Vorhandensein des gesuchten Objektes in der Liste Aufschluss gibt. Das übergebene Suchkriterium wird über den erwähnten Lambda-Ausdruck abgekürzt. Mit **n** ist es hierbei möglich direkt auf ein Feld in der Liste zuzugreifen und diese Abzufragen (**n.Contains**) ob sich ein bestimmter Ausdruck darin befindet. Sollte keine der

beiden inneren Schleifen unterbrochen werden, stimmen alle Portkommandos überein und die Funktion gibt ein **true** zurück.

5.3.2 Runnable zu Task Mapping

Wie bereits im Konzept erwähnt, sind die Informationen über das Task Mapping nicht direkt zu erschließen. Deshalb wird die entsprechende RTE-Quellcode-Datei mit Hilfe des RTE-Checkers (siehe 5.3) untersucht. Mit einer Liste vom Typ **RunnableTaskMapping** werden die Beziehungen zwischen Task und Runnable gesammelt. Dieser Datentyp speichert den Runnable-Namen, den korrespondierenden Task-Namen und wird später durch das Timing (Zeitlicher Aufruf der Runnables) ergänzt. Somit stehen für die Simulation alle benötigten Scheduling-Parameter bereit. Nach AUTOSAR-Spezifikation [AUT14h] werden in der **Rte.c** Datei die entsprechenden Taskbodys generiert, welche nach erfolgreicher Kompilierung vom Scheduler des Betriebssystems ausgeführt werden. Der Algorithmus sucht zu erst den Namen der Runnable, welcher über das betreffende SWC-Objekt abgerufen werden kann und prüft dann in welchem Taskbody sich diese befindet.

```
int nr = lines.FindIndex(p => p.Contains(runnable));
for (int i = nr ; i > 0 ; i--)
{
   if (lines[i].Contains("TASK("))
   {
      return lines[i];
   }
}
```

Listing 5.7: Algorythmus zum Suchen nach Runnable-Task-Beziehungen

Wie im Listing zu sehen, wird zuerst die Zeilennummer gesucht, in welcher sich die gesuchte Runnable befindet. Auch hierbei wird wieder *Linq* Technologie verwendet, um eine Suche die normalerweise mehrere Zeilen Code in Anspruch nimmt, zu vereinfachen. Mit Hilfe eines Lambda-Ausdrucks (p => p)wird das gesuchte Objekt zurückgegeben. Mit der Kenntnis über die Zeilennummer muss der Zähler solang dekrementiert werden, bis der umschließende Task gefunden wurde.

5.3.3 Zuweisen von RTE-Dateien

Die Bereitstellung von Testdaten an den entsprechenden Schnittstellen, erfordert die Kenntnis über die RTE-Dateien in denen diese implementiert wurden. Zur Konfigurationszeit soll eine Softwarekomponente neben den zur Kommunikation benötigten Ports und deren Kommandos, die erforderlichen Dateien „kennen“. Anhand dieser Informationen ist der RTE-Datei-Generator in der Lage die entsprechenden Vorkehrungen zu treffen, um so während der Simulation die definierten Testdaten übergeben zu können. Die genau Funktionsweise wird im Kapitel 5.6 RTE-File-Generator genauer beschrieben.

5.4 Visualisierung von Systeminformationen

Systeminformationen sollen dem Nutzer schnell zur Verfügung stehen. Das betrifft nicht nur die grafische Darstellung der einzelnen Komponenten wie Softwarekomponenten und deren Verbindungen, sondern auch Informationen zu diesen als Baumstruktur.

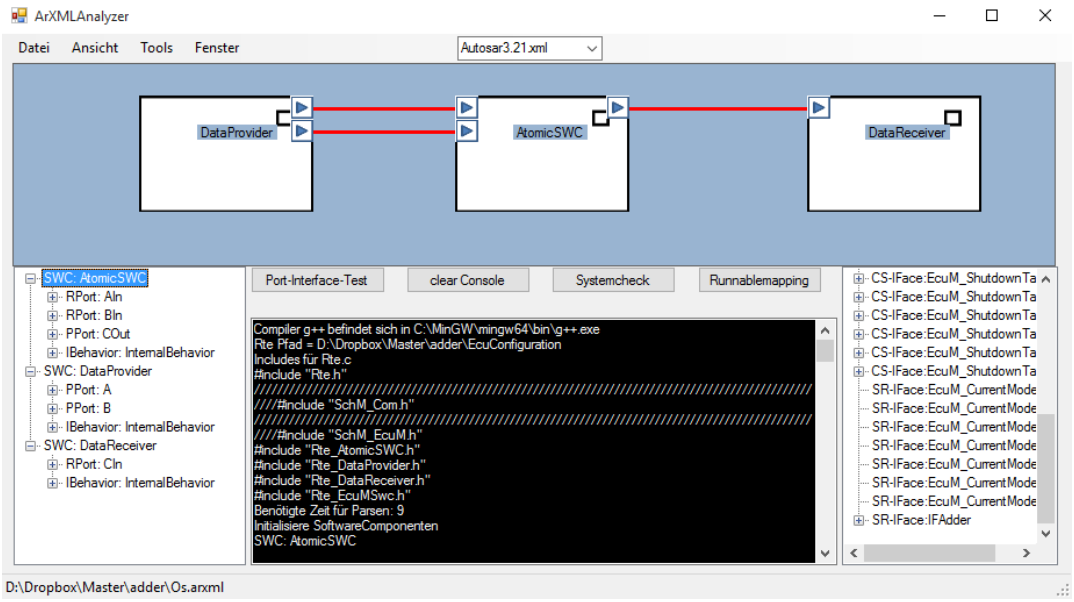

Abbildung 5.2: Hauptfenster mit Visualisierer

Wie in Abbildung 5.2 zu sehen, werden Komponenten und deren Unterkomponenten als Baumstruktur dargestellt. Das heißt die zu den SWCs zugehörigen Ports und interne Verhalten, können wiederum Eigenschaften darstellen, indem auf das entsprechende Plus gedrückt wird. Somit erhält der Nutzer alle systemrelevanten Informationen auf einen Blick. Diese Ansicht (**TreeView**) wird dem Nutzer über das Hauptfenster (siehe Abbildung 5.2) präsentiert. Außerdem wird bei jedem Port in der Baumansicht, dass zuvor generierte und dem einzelnen Port zugewiesene Portkommando angezeigt.

5.4.1 Menüleiste

Die Menüleiste (Abbildung 5.2) bietet dem Nutzer mit den Menübutton die Möglichkeit Programm- und Testeinstellungen vor zu nehmen. Über das **Dateimenü** kann eine ARXML-Datei importiert oder das gesamte Projekt gespeichert und geladen werden. Das Datei-Menü bietet folgende Möglichkeiten.

XML Importieren

Diese Option ist die erste Anlaufstelle, wenn eine Applikation getestet werden soll. Mit Hilfe eines Windows-üblichen Dateiexplorers kann eine oder mehrere XML-Dateien geladen werden. Anschließend werden alle geparsten Informationen präsentiert.

Projekt Speichern und Laden

Mit diesen Optionen wird das gesamte Projekt samt Einstellungen persistent auf die Festplatte abgelegt. Eine einfache Umsetzung bietet hierfür das .NET-Framework mit der Binären Serialisierung ([Küh13]). Diese wird statisch in der Klasse *PersistentOperation* implementiert, welche auch für weitere persistente Operationen verantwortlich ist. Werden die Daten serialisiert, geschieht dies über einen Byte-Stream, der die Daten binär ablegt. Ferner wird eine typisierte Datenspeicherung realisiert. D. h. werden Informationen eines bestimmten Typs abgelegt, können auch nur diese wieder in das benötigte Objekt deserialisiert werden ([Küh13]).

Tools

Über die Option „**Tools**“, kann das Testkonfigurationsfenster (5.5, das Pfadmenü, der Profilmanagers (5.2.3) und der XML-Validierungsdialog gestartet werden.

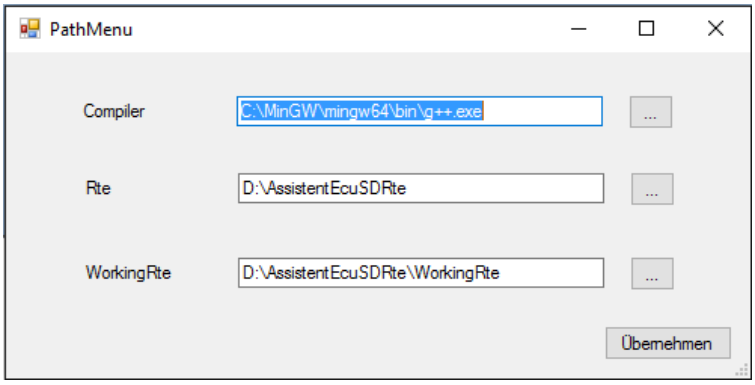

Abbildung 5.3: Pfadmenü

Im Pfadmenü kann der Ort der zu importierenden RTE und des zu verwendenden Compilers festgelegt werden. Der Compiler-Pfad wird während der Initialisierungsphase beim Programmstart geprüft und über die Konsole ausgegeben. Wurde kein Compiler gefunden, kann der Pfad angepasst werden.

Im Menü **Fenster**, kann der Inhalt des Visualisiers in ein größeres Fenster ausgelagert werden. Dies ist brauchbar, wenn viele Komponenten angezeigt werden müssen. Wird das Fenster geschlossen, wird der Inhalt dem Standard Fenster zurück gegeben.

5.4.2 Visualizer

Die gezeichnete Systemdarstellung der Komponenten wird mit Hilfe des Visualizers realisiert. Dieser ist nicht nur ein Kontainer für die Komponenten, sondern vielmehr ein Paket aus mehreren Elementen wie dem VisualPanel, einem Controller für die Logik, und den anzuzeigenden Elementen wie SWCs, Ports, Runnables und Verbindungen. Erbt von UserControl, einer C spezifischen Komponente für alles was mit Benutzerinteraktion zu tun hat.

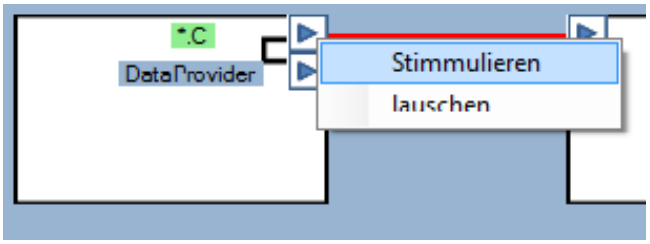

Abbildung 5.4: Kontextmenü eines Ports

Obige Abbildung zeigt die Möglichkeit Ports für eine Stimulation oder das Lauschen von Signalen fest zulegen. Hierbei kann der Nutzer jeden beliebigen Port mit beiden Kontextmenüoptionen belegen. Dieser Ansatz zieht mehrere Testvarianten nach sich, die es dem Akteur ermöglichen durch geeignete Auswahl von Ports, einen Unit-Test, mehreren SWCs im Verbund oder einen Test am Gesamtsystem durch zuführen. Diese Methodik erlaubt es, an einen Empfängerport ein Signal anzulegen und an einem Sendeport auszulesen. Wird ein Port stimuliert wird dieser Rot markiert und bei der Option lauschen ist die Markierung Grün. Somit ist der Akteur jederzeit im Bilde, welchen Status die einzelnen Ports aufweisen.

5.4.3 Systeminformations- und Fehlerausgabe

Abbildung 5.2, zeigt die Textkonsole, die zentral jegliche Arten von Informationen anzeigt, um so ein Feedback auf ausgeführte Operationen zu geben. Das sind vor allem Fehler oder Unstimmigkeiten, können aber auch Systeminformationen sein. Realisiert wird die Ausgabe durch eine Textkonsole, ähnlich der Eingabeaufforderung in Windows oder der Shell in Linux, nur vorerst ohne Befehle. Die Konsole wird durch eine Textbox die über mehrere Zeilen verfügt, präsentiert.

```
public void writeToConsole(string text)
{
    mainForm.tB_Console.AppendText(text + "\r\n");
}
```

Listing 5.8: SWC Ausschnitt aus einer ARXML

Mit Hilfe der Funktion writeToConsole im Controller, soll jedes Objekt welches relevante Systemoperationen durchführt, die entsprechenden Meldungen anzeigen können. Das obere Listing zeigt das Anhängen (**AppendText**) der Textzeilen, welche der Methode beim Aufruf übergeben werden. Weiterhin kann über die Funktion ClearConsole (nicht im Listing) die gesamte Textbox geleert werden. Um die Konsole aus anderen Klassen ansprechen zu können, muss normalerweise nur die Instanz vom Controller übergeben und die Funktion aufgerufen werden. Da drei Programmteile als Klassenbibliotheken implementiert wurden, dürfen zu diesen keine Abhängigkeiten bestehen, also vor allem keine Instanzen übergeben werden, da diese in einer Klassenbibliothek nicht bekannt sind. Das .Net-Framework bietet für die Entkopplung von Objekten unter anderem Events an. Mit diesen können Nachrichten an alle Objekte gesendet werden, die sich dafür angemeldet haben.

```
rteChecker.NewConsoleMessageReceived += ConsoleMessageReceived;
 ...
private void ConsoleMessageReceived(object sender, string e)
{
    writeToConsole(e);
}
```

Listing 5.9: SWC Ausschnitt aus einer ARXML

Das obere Listing zeigt die Anmeldung des Events (**NewConsoleMessageReceived**) am Beispiel des RTE-Checker am Controller. Das RTE_Checker-Modul implementiert hierfür das Interface IWriteToConsole und zwingt somit alle Teilnehmer die Methode OnNewConsoleMessageReceived zu verwenden, um so Nachrichten für die Konsole bereitzustellen.

5.5 Test Konfigurations-Panel

Die Generierung der Testumgebung, erfordert einige zuvor konfigurierte Parameter. Um diese Parameter vom Benutzer festlegen zu lassen, wird das Konfigurationsfenster über den Punkt SWC-Test in der Menüleiste (5.4.1) gestartet werden.

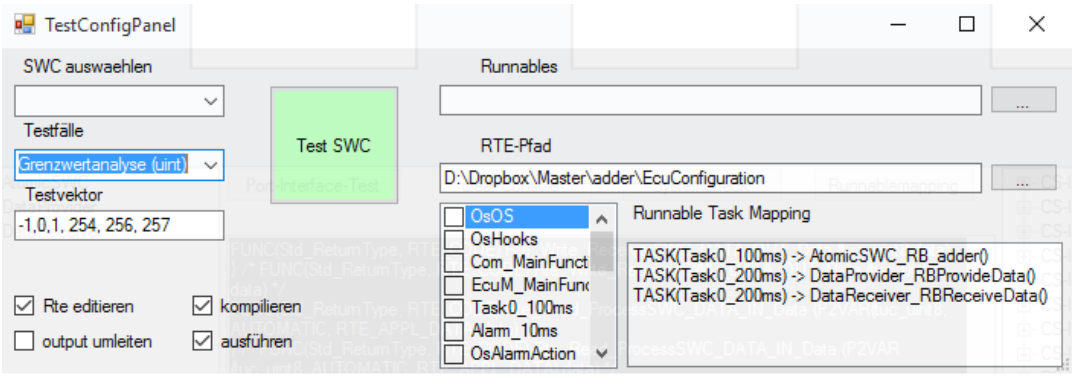

Abbildung 5.5: Test-Konfigurationsfenster

Die Abbildung zeigt die einzelnen Auswahl- und Einstellmöglichkeiten. Im Feld **Testfälle** kann ein gewünschter Testfall ausgewählt werden. Bisher beschränkt sich diese Auswahl auf eine Grenzwertanalyse, bei der der minimale und maximale Wert einer Variablen getestet wird und einen eigen definierbaren Vector der im Feld darunter (Abb. 5.5) initialisierbar ist. Unten links befinden sich checkboxen. Ist **Rte editieren** ausgewählt, wird die RTE für jeden Test neu angepasst. Diese Option ist Standard und wird immer benötigt, wenn sich Parameter zum Testen ändern (z. B. andere Testwerte). Mit **output umleiten** wird die Möglichkeit geboten, alle Meldungen über das Command-Fenster von Windows aus zugeben. Soll die Simulation nicht jedes mal neu kompiliert werden, wenn Beispielsweise Tests mehrmals gestartet werden, kann die Option **kompilieren** deaktiviert werden. Wird eine fertig kompilierte Applikation für andere Zwecke benötigt, muss diese nicht extra ausgeführt werden. Dies wird erreicht, indem **ausführen** deaktiviert wird. Auf der rechten Seite, ist die Auflistung aller Tasks im Projekt zu sehen. Per Auswahl über checkboxen soll später mal, das Starten von Diensten möglich sein, welche normalerweise nicht direkt in der Applikation benötigt werden. Dazu müssen diese Dienste als Dummy oder Mock (2.2.3) generiert werden, um so auf die Funktionsaufrufe reagieren zu können. Im Feld darüber befindet sich die Anzeige zum RTE-Arbeitsverzeichnis. Sollte der Pfad nicht korrekt sein, kann dieser über einen Pfad-Navigator angepasst werden.

5.6 Generierung der Testumgebung

Die Generierung der Testumgebung erfordert nicht nur die Informationen, welche aus dem Import gewonnen werden, sondern auch die Konfiguration des Gesamtsystems und das Hinzufügen benötigter Ressourcen. Die wichtigste Komponente für die Erstellung der Simulationsumgebung ist der RTE-File-Generator. Bevor dieser erläutert wird, müssen zuerst die entsprechenden „Werkzeuge„ Vorgestellt werden, die zur Abarbeitung der Befehle benötigt werden. Diese sind der Code-Generator und die CommandList.

RTE-File-Generator

Wie im Konzept angesprochen werden für eine Kompilierbare und Lauffähige Simulationsumgebung ergänzende Quellcode-Dateien benötigt. Das sind zum einen Dummy-Header (siehe 5.6.1) für fehlende Includes und zum anderen Dateien, die Testdaten und den scheduler zur Verfügung stellen. Dies wird mit Hilfe des RTE-File-Generators ermöglicht.

Code-Generator

Als Herzstück des RTE-File-Generators hat der Code-Generator die Aufgabe alle Quelltext-Dateien zu erzeugen. Außerdem stellt er wiederkehrende Abarbeitungsroutinen wie z. B. Methodenrümpfe oder das Einbinden von Header-Dateien bereit. Mit der Funktion createFile wird die gewünschte Datei erstellt. Der Übergabeparameter ist die CommandList. Jede zu generierende Zeile Code wird in dieser Liste abgelegt. Der gesamte Inhalt einer Code-Datei wird somit durch die Liste dargestellt und als Datei gespeichert.

5.6.1 Stub und Dummy-Header

Dieses Unterkapitel beschreibt die eigentliche Trennung zur Basissoftware. Weil das Testobjekt für eine Ausführung auf andere Komponenten angewiesen ist, müssen diese wie im Grundlagenkapitel 2.2.3 erklärt, als Dummy bereitgestellt werden. Damit der RTE-File-Generator weiß, welche Dummy-Header-Dateien zu erstellen sind, werden alle nicht vorhandenen nach Spezifikation geforderten Dateien erzeugt. Das Projekt bietet aber auch Stubs wie ComDummy und StateManagerDummy. So wird z.B. ein Com-Modul Stub generiert, welcher die Funktionen Com_ReseiveSignal, Com_SendSignal, Com_MainFunktionRx und Com_MainFunktionTx des Moduls präsentiert und nur aus gibt, dass diese aufgerufen wurden. Das gleiche gilt für den StateManger-Stub mit der Funktion EcuM_MainFunktion. Auch die entsprechenden Header-Dateien zu diesen werden automatisch generiert, um alle Funktionen in der

Simulation bekannt zu machen. Weiterhin wird der benötigte Header Os.h bereitgestellt, welcher das folgende sehr wichtige Macro liefert.

```
#define TASK(x) void OS_TASK_##x(void)
```

Mit dieser Zeile wird es dem Taskverwaltungsmodul ermöglicht, generisch alle Task-Körper im RTE-Modul anzusprechen.

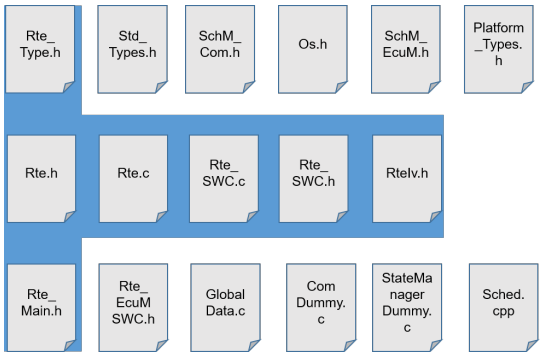

Abbildung 5.6: Abhängigkeiten des Testobjekts

Die Abbildung zeigt die an einer Applikation beteiligten Dateien. Diese werden benötigt um eine Applikation mit all ihren Bestandteilen zu erstellen. Die Dateien auf blauem Hintergrund repräsentieren den originalen RTE-Code. Die anderen die vom RTE-File-Generator erzeugten Dummys.

5.6.2 Bereitstellung externer Informationen

Die Übergabe der Testdatensätze wird wie bereits erwähnt mit einem Datenvektor realisiert. Dazu wird eine Datei namens *GlobalData.c* generiert, welche die Testdaten und den globalen Iterator *counter* bereit stellt. Der Zähler wird mit 0 initialisiert und dient dem iterieren aller Felder im Testdaten-Vektor. Die Globalisierung dieser Variablen ermöglicht den Einsatz an jeder Schnittstelle in der RTE, das heißt in jeder mit zu kompilierenden Datei. Bekannt gemacht werden diese Informationen in der generierten Headerdatei *Std_Types.h*. Das hat den Vorteil, dass in keiner anderen Datei ein extra Verweis generiert werden muss, da dieser Header schon überall bekannt ist. Dieses Herangehen, trägt unter anderem dazu bei, dass das Testobjekt nicht verändert werden muss.

5.6.3 Task-Verwatungsmodul generieren

Im Konzept wurde aufgezeigt, dass für das Ausführen der Runnable-Entitys eine eigens Implementierte Lösung entwickelt wird. Diese beherbergt die eigentliche Basis

zur Programmausführung, nach der fertigen Kompilierung. Das Task-Verwatungsmodul wird in C++ 11 implementiert. Dies hat den Vorteil, dass nicht Unmengen Quellcode generiert werden müssen [Gri13]. Außerdem findet hier die eigentliche Kapselung von der Basissoftware statt. Es wird nochmals darauf hingewiesen, dass es sich hierbei nicht um einen richtigen Scheduler handelt, wie es bei dem entsprechenden Basissoftwaremodul der Fall ist, denn das würde den Rahmen dieser Arbeit sprengen. Vielmehr wird ein möglicher Weg aufgezeigt, wie Runnables betriebssystemunabhängig aufgerufen und somit die Applikation ausgeführt werden kann. Wie im Grundlagenkapitel erklärt, generiert der RTE-Generator so genannte Taskbodys in der RTE-Quellcode-Datei `Rte.c`. Da sich in diesen die Runnable-Funktionsaufrufe befinden, werden bei dem Ausführen von Taskbodys auch die Runnables ausgeführt. Hierbei muss auch nicht darauf geachtet werden, in welcher Reihenfolge dies geschieht, da diese der entsprechende RTE-Generator bereits festgelegt hat.

Runnable-Trigger-Funktionen

Das Task-Verwatungsmodul wird durch die Datei Sched.cpp repräsentiert. Diese wird mit Hilfe des RTE-File-Generators erzeugt. Da diese Datei später als Teil des Projekts mit Kompiliert werden muss oder vielmehr das Herzstück der Simulationsumgebung darstellt, werden als erstes alle benötigten Systembibliotheken und anschließend die Datei Rte.h durch inkludieren bekannt gemacht. Hierbei ist die C Bibliothek chrono (<chrono>) zu erwähnen, welche das zeitliche Verhalten der Runnables realisiert, sofern diese durch ein Timing-Event getriggert werden. Für jede erkannte auszuführende Runnable wird eine Taskfunktion generiert, die den eigentlichen Taskbody in der Basisdatei (Rte.c) ausführt. Des weiteren werden die Generierten Taskbodys für eine variable Benutzung bekannt gemacht. Das Abarbeiten paralleler Tasks wird durch Thread Objekte aus der Bibliothek „thread“ ermöglicht.

Die Main-Funktion

Wie in den gängigsten Programmiersprachen auch, wird für die Ausführung eines Programms eine Main-Funktion benötigt, welche die Programmsteuerung übernimmt ([Rol07]). Wie Eingangs erwähnt, stellt sched.cpp die Hauptdatei dar, weswegen hier auch die Main-Funktion zum Einsatz kommt. Laut Spezifikation ([AUT14h]) wird die RTE-Startphase mit der Funktion Rte_Start() eingeleitet. Dabei werden alle Signale und Interrunnable-Variablen initiiert und bei erfolgreichen Abarbeiten ein OK- Wert zurück gegeben. Anschließend folgt eine Schleife deren Durchlauf sich an der Länge der Testdatensätze orientiert, um so alle Werte aus dem Vektor übergeben zu können. Im Schleifenkörper wird für jeden Task ein Thread-Objekt erzeugt und der auszuführende Task übergeben. Folglich wird der thread gestartet und die Zähl-Variable (Counter) iteriert, um so den Testdatenvektor Feld für Feld abzufragen.

5.7 Kompilierung der Applikation und Testumgebung

Der entscheide und finale Schritt zur Fertigstellung einer Ausführbaren Applikation ist die Kompilierung aller zum Projekt gehörigen Quellcode-Dateien. Hierzu müssen alle in dieser Arbeit vorhergehenden Schritte erfolgreich verlaufen sein. Es müssen alle Dateien mit den zugehörigen Funktionen vorhanden sein, denn der Compiler prüft vorab mit dem Pre-Prozessor alle eingebundenen Dateien und das Vorhandensein aller zu verwendenden Funktionen und Variablen.

Kompilieren mit MinGw

MinGw steht für **Minimalist GNU for Windows** [Shp13] und bietet den Vorteil, dass der Compiler immer zum Einsatz kommen kann, da sich die freie Verwendung nach dem GNU-Modell für solche Zwecke anbietet. Das MinGw Packet besteht aus mehreren Compilern. Für die Testumgebung wird der 64bitige g++ verwendet, welcher speziell für C++ Programme entwickelt wurde. Dem ArXmlAnalyzer muss vorher der Pfad zum Compiler bekannt sein. Ist dies nicht der Fall, kann der Aufenthaltsort über das Pfadmenü bekannt gemacht werden. Die Übersetzung in Maschinencode erfolgt durch den Aufruf des Compilers mit der Option c und dem Pfad zur Quellcodedatei. Ein konkreter Aufruf sieht dann folgender Maßen aus.

```
g++ -c -std=c++11 sched.cpp
```

Der erste Teil der Anweisung (g++) gibt an welcher Compiler verwendet werden soll. Mit -std=c++11 wird der Namespace auf die Version 11 von C++, zum Schluss wird die zu kompilierende Datei angegeben.

Batchdatei

Für die Kompilierung wird ähnlich wie bei *make* eine Batchdatei (*compileRte.bat*) mit Compiler-Befehlen erzeugt. Somit wird es möglich bei eventuellen Fehlern oder Parametererweiterungen für den Compiler selbst Hand anzulegen und den Kompiliervorgang zu beinflussen. Mit der Methode *createBatchFileCompile* bietet der Codegenarator die Möglichkeit, eine ausführbare Datei zu erzeugen. Dazu wird der Referenzpfad eines SWC-Objektes verwendet, wie im Kapitel sec:ZuweisenVonRTEDateien beschrieben. Des weiteren werden alle Standardisierten und erzeugten RTE-Dateien der Befehlsliste mit übergeben.

```
Compiler Pfad\g++.exe -c RTE Pfad\Rte.c
Compiler Pfad\g++.exe -c RTE Pfad\RteIV.c
Compiler Pfad\g++.exe -c RTE Pfad\GlobalData.c
Compiler Pfad\g++.exe -c RTE Pfad\Rte_SWC.c
Compiler Pfad\g++.exe -c RTE Pfad\Runnable.c
Compiler Pfad\g++.exe -c -std=c++11 RTE Pfad\sched.cpp
Compiler Pfad\g++.exe -o Rte *.O
```

Listing 5.10: Compiler-Befehl

Listing 9 zeigt den Inhalt der generierten Batch-Datei. Dabei ist noch einmal gut zuerkennen, welche Dateien für den Simulator und die Applikation benötigt werden. **Rte.c** und **RteIV.c** sind die schon vorhandenen RTE-Dateien, welche lediglich in das Verzeichnis *workingDirectory* kopiert werden müssen. **GlobalData.c** stellt alle Benötigten Informationen bereit (siehe 5.6.2) und **Rte_SWC** repräsentiert die Schnittstellen einer Softwarekomponenten, wobei „SWC“ für den Namen der Komponente steht. Die eigentliche Verarbeitungslogik (**Runnable.c**) einer SWC, darf hierbei auch nicht fehlen. Anschließend folgt der die Übersetzung des Schedulers (**sched.cpp**). Wurden alle Dateien kompiliert liegen diese als Objekt-Dateien (*.o) vor. Diese werden Abschließend mit dem Befehl -o gelinkt und zu einer ausführbaren Datei verarbeitet. Für das anschließende Ausführen des Programms wird eine weitere Batch-Datei generiert.

5.8 Architektur des ArXmlAnalyzer

Die Implementierung des ArXmlAnalyzer orientiert sich an dem MVC Muster. Für die einzelnen im Konzept erwähnten Phasen, wird jeweils eine Komponente bereit gestellt, die für die entsprechende Anforderung eine Lösung bietet.

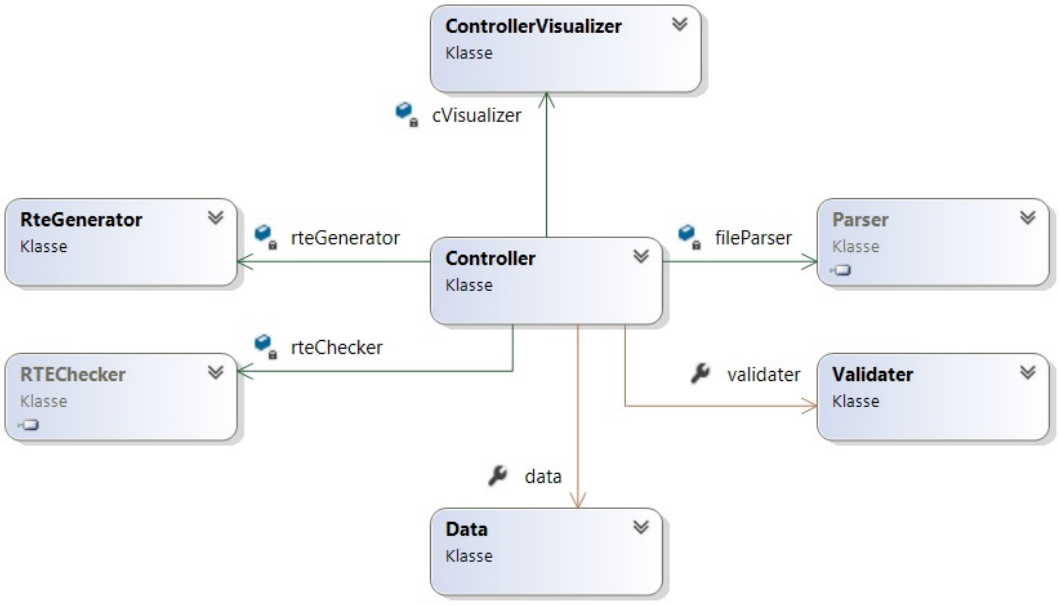

Abbildung 5.7: Programmaufbau

Die Abbildung zeigt die gesamte Struktur des Programms. Hierbei ist zu erkennen das es sich um mehrere Komponenten handelt. Dies hat den Hintergrund, dass alle

zu erfüllenden Anforderungen Modular behandelt werden. Die Teile die sich direkt am Controller befinden sind normale Packages mit den entsprechenden Klassen. Die durch Pfeile Verbundenen Teile werden als Klassenbibliotheken bereitgestellt. Diese ermöglichen das einfache Einbinden in andere Projekte und geben so einem anderen Entwickler die Möglichkeit, bereits implementierte Lösungen wieder zu verwenden oder zu erweitern. Die einzelnen Programmteile wurden in den entsprechenden Unterkapiteln näher erläutert. Hier ist nochmal zu erwähnen das die Kommunikation mit dem Kontroller Event-Gesteuert ist und somit keine Abhängigkeiten vorhanden sind.

5.9 Zusammenfassung

Mit der Implementierung des im Konzepts vorgeschlagen Lösungsweg, wurde bewiesen, dass die einzelnen Phasen im Projekt realisierbar sind. Alls Beweis diente eine in SystemDesk erstellte Applikation, welche den Input für das Projekt bereitstellt. Die Validierung der ARXML wurde mit der Hilfe des .Net-Frameworks in wenigen schritten realisiert. Der Datenimport wird durch einen Parser realisiert, der in der Lage ist, direkt Objekte zu erzeugen und diese als Liste zurückzugeben. Durch die Implementierung des Profilmangers können auch andere AUTOSAR-XML-Dateien geparst werden, ohne dass hierfür der Quellcode angepasst werden muss. Profile können zur Laufzeit ausgewählt und eingesetzte werden. Die im Konzept theoretisch vorgeschlagenen Analyseverfahren der RTE-Quellcode-Dateien, wurden mit Hilfe des implementierten RTE-Checkers umgesetzt. Neben der Visualisierung aller relevanten Komponenten einer Applikation, gibt es eine zentrale Konsole die den Nutzer über alle Tätigkeiten und Fehler während dem Betrieb informiert. Dem Nutzer wird durch das einfache Anklicken von Ports die Möglichkeit geboten einen Port zu stimulieren oder diesen auszulesen. Die Generierung der Testumgebung konnte ebenfalls bewiesen werden, in dem eine auszuführende Datei erzeugt wurde, die überall ausgeführt werden kann. Mit Hifle von Batch-Dateien kann ein Nutzer in den Kompilierungsprozess eingreifen und neue Module für die Applikation bereitstellen. Auch die Funktionalität des selbst implementierten Task-Verwaltungsmodul konnte durch die Ausführung aller Runnables bewiesen werden.

6 Testergebnisse

Ein wichtiger Baustein bei der Softwareentwicklung stellt das Testen dar. Im Konzept wurde darauf hingewiesen, dass sich diese Arbeit in das statische Analysieren und dynamische Testen unterteilt. Daher werden diese zwei Bereiche getrennt betrachtet. Alle Testobjekte wurden mit SystemDesk erstellt.

6.1 Statische Analyse

Für die statische Analyse dient die Applikation Adder aus dem Konzept. Dabei sind die generierten ARXML- und RTE-Dateien relevant. Die einzelnen Prüfungen werden in der gleichen Reihenfolge wie im Konzept absolviert. Da mit SystemDesk eine fehlerfreie Applikation erzeugt wurde, wird zum Testen die gleiche nochmal mit Fehlern erstellt. Diese wird so manipuliert, dass die AUTOSAR-Spezifikationsvorgaben verletzt werden.

6.1.1 Vorbereitung

Zuerst wird die Applikation Adder mit SystemDesk neu erstellt. Dabei werden die möglichen, im Konzept (4.3.1) erwähnten, zu prüfenden Fehler eingebaut.

Außerdem wird eine **ARXML mit Version 4.2** der Adder-Applikation mit SystemDesk erstellt. Diese soll die Funktionalität des Profilmanagers 5.2.3 unter Beweis stellen. Da der Profilmanager eine XML-Profil-Datei importiert, wird diese vorher erstellt. Dazu werden alle Änderungen der Tag-Namen (siehe [AUT14g]) berücksichtigt und in der neuen Datei angegeben.

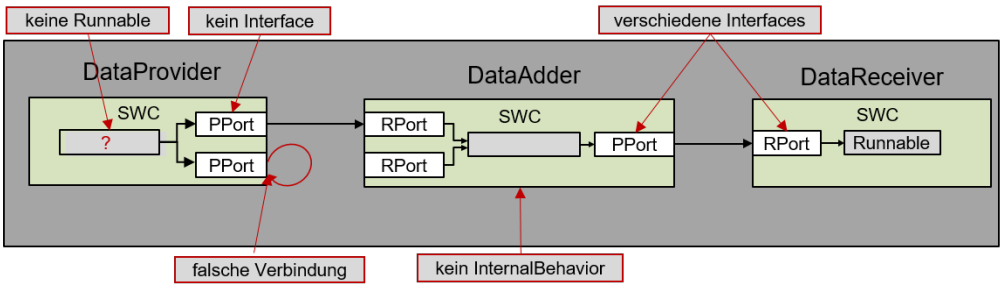

Abbildung 6.1: Applikation mit Fehlern

Abbildung 6.1 zeigt die eingebauten Fehler in der Applikation Adder. Diese müssen beim Laden vom ArXmlAnalyzer erkannt und ausgegeben werden.

VFB003: Für das Interface wird die Regel, dass zwei miteinander verbundene Ports das gleiche Interface aufweisen müssen, verletzt. Dazu wird dem PPort vom DataProvider kein und dem RPort vom DataReceiver ein anderes Interface zugewiesen.

VFB043: Wie in der Abbildung zu sehen, wird dem DataProvider die Information zur Runnable entzogen.

VFB010: Die Regel, dass ein PPort exakt mit einem RPort verbunden sein muss, wird an dieser Stelle verletzt. Im Bild wird dies durch den mit sich selbst verbundenen Port dargestellt.

Vom Autor definierte Fehler

Die zu überprüfenden Fehler reichen nicht aus, um eine laufende Simulationsumgebung zu garantieren. Zu weiteren Tests werden daher vom Autor selbst Fehler definiert.

Internal Behavior Das Internal Behavior (IB) ist mit eine der wichtigsten Voraussetzungen für die spätere Simulation, denn hier wird das gesamte Verhalten einer Softwarekomponente beschrieben. Für die Überprüfungen wird deshalb der SW-C Adder kein IB zugewiesen.

Input-Vergleich und Taskmapping

Wie im Konzept (4.3.2) angesprochen, werden neben der Objektprüfung auch RTE-Dateianalysen durchgeführt. Somit kann z. B. sichergestellt werden, dass die XML auch zur RTE gehört. Da bei dieser Technik die Portkommandos aus dem RTE-Modul ausgelesen werden, wird bei einer SW-C lediglich der Name (Shortname) geändert. Dies würde darauf hinweisen, dass beide Input-Objekte nicht korrespondieren. Weiterhin kann aufgrund der RTE-Analyse das Taskmapping rekonstruiert werden. Dies soll nur auf Richtigkeit geprüft werden, da sonst Fehler in die Code-Dateien geschrieben werden müssten.

Visualisierung der Komponenten

Die Visualisierung der Komponenten kann bei jedem der oben erwähnten Tests mit überprüft werden. So würde bspw. keine Runnable angezeigt, wenn diese nicht zugewiesen würde. Weiterhin würde bei einer falschen Portverbindung die Verbindungslinie anders aussehen.

6.1.2 Parsen der AUTOSAR 4.2 ARXML

Wie im Konzept 5.2 erwähnt, sollen auch andere AUTOSAR-Versionen einlesbar sein. Dazu wird die Applikation Adder in SystemDesk 4.2 und somit eine andere Version der ARXML erstellt. Im Abschnitt 5.2.3 Profilmanager wurde außerdem aufgezeigt, wie eine Profil-Datei aussieht und wie einfach diese erstellt werden kann. Für den Test wird so eine Datei mit einem einfachen Texteditor auf Basis der Version 3.2 erzeugt. In dieser werden alle zu ändernden Tag-Namen von AUTOSAR 4.2 bekannt gemacht.

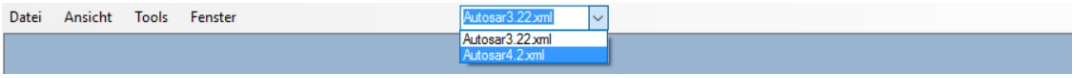

Abbildung 6.2: Profilauswahl

Diese Profildatei wird dann über den Profilmanager importiert und kann zur Laufzeit wie in Abbildung 6.2 zu sehen, ausgewählt werden. Anschließend ist es möglich, die mit SystemDesk erstellte ARXML zu lesen. Hierbei ist es wichtig, dass ebenfalls alle Eigenschaften der Applikation dargestellt werden.

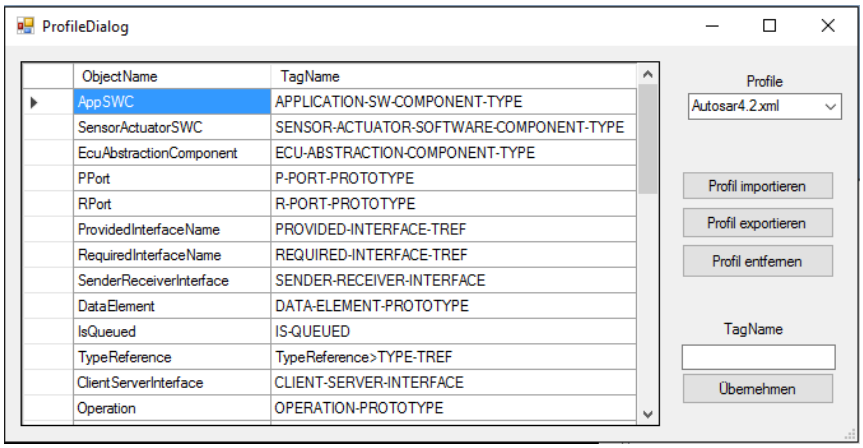

Abbildung 6.3: ProfilDialog

Die Abbildung zeigt den Profildialog mit Profilmanager. Hier wird für das Testszenario über den Button **Profil importieren** die Profildatei „**Autosar4.2.xml**“ eingelesen. Der Inhalt der Datei wird in der Tabelle dargestellt. Sobald ein Profil importiert wurde, steht es zum Parsen bereit (siehe Abbildung 6.2).

6.1.3 Testergebnisse

Die AUTOSAR-XML wird über das Dateimenü „importieren“ geladen. In der Mitte der Menüleiste (Abbildung 6.2) kann die zu parsende AUTOSAR-Version eingestellt werden. Der Pfad zu den RTE-Dateien kann über das Menü „Pfade“ angepasst werden. Mit dem Knopf „Systemtest“ werden alle implementierten Spezifikations- und die vom Autor selbst definierten Vorgaben geprüft. Dabei dient die Konsole (5.4.3) als zentrale Ausgabe aller Informationen.

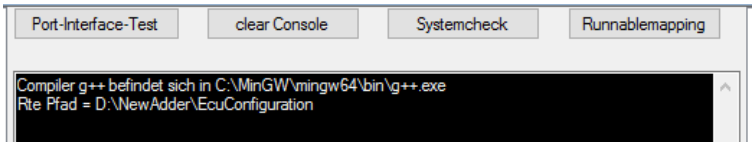

Abbildung 6.4: Konsole

Beim Starten des ArXmlAnalyzer gibt diese z. B. gleich Auskunft über den aktuellen RTE-Pfad und ob der Compiler gefunden wurde. Die Testergebnisse werden anhand einer Tabelle dargestellt. Dabei wird zuerst der erzeugte Fehler und anschließend ein Soll- und Ist-Zustand angegeben.

Fehler	Soll (Ausgabe)	Ist (Ausgabe)
Kein Interface an PPort	Port: B hat kein Interface	Port: B hat kein Interface
Verschiedene Interfaces an PPort von Adder und RPort von Receiver	PPort und Rport haben verschiedene Interfaces	PPort und Rport haben verschiedene Interfaces
Keine Runnable in DataProvider	DataProvider hat keine Runnable	DataProvider hat keine Runnable
Falsch verbundener PPort	Port: B hat kein Interface	Port: B hat kein Interface
Kein InternalBehavior bei DataAdder	DataAdder hat kein InternalBehavior	DataAdder hat kein InternalBehavior

Tabelle 6.1: Übersicht der Ergebnisse

Bei der Darstellung aller relevanten Systeminformationen (siehr 5.4.3) konnte sowohl die ARXML 3.2 als auch 4.2 eingelesen und dargestellt werden. Die Ergebnistabelle zeigt, dass alle eingebauten Fehler durch die statische Analyse gefunden wurden. Es ist zu beachten, dass es auch Testfälle geben kann, die noch nicht berücksichtigt wurden. Auch die unterschiedlichen AUTOSAR-Versionen konnten mit Hilfe des Profilmanagers eingelesen und angezeigt werden.

6.2 Dynamisches Testen

Bei diesem Test wird der Prüfling in der generierten Simulationsumgebung ausgeführt. Die Grundlagen zum Thema sind dem Kapitel 2.2.2 Dynamischer Test zu entnehmen. In dieser Arbeit geht es in erster Linie darum, eine Applikation auf ordnungsgemäße Ausführung zu prüfen, um so die Richtigkeit des Simulators zu beweisen. Außerdem soll demonstriert werden, wie einfach das Stimulieren und Abhören der Schnittstellen realisiert wurde. Dem Benutzter werden nach dem Test zum einen eine Konsole und zum anderen ein Diagramm präsentiert. Die Konsole zeigt zur Laufzeit die Ausgabe der Werte, die an den selektierten Ports anliegen. Am Ende des Tests wird ein Gnuplot-Skript generiert, welchem alle benötigten Ergebnisse übergeben werden. Dies wird in Form eines Zeit-Wert-Diagramms dargestellt.

6.2.1 Vorbereitung

Für den Simulationstest wird eine neue Adder-Applikation mit SystemDesk realisiert. Dabei ist vor allem das zeitliche Verarbeiten der Testdaten relevant. Denn somit kann auch das Taskverwaltungsmodul auf ordentliche Abarbeitung der Tasks geprüft werden.

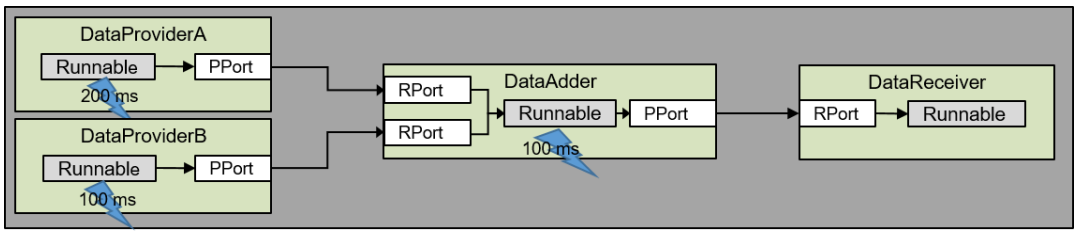

Abbildung 6.5: Applikation Adder

DataProviderA Wie in Abbildung 6.5 zu sehen, besitzt diese SW-C eine Runnable-Entity, die alle 200 ms aufgerufen werden soll. Diese implementiert einen Zufallszahlengenerator und ist mit dem PPort verbunden.

DataProviderB Die Applikation bekommt eine weitere Softwarekomponente (**DataProviderB**). Die zugehörige Runnable übergibt alle 100 ms einen statischen Wert an den Ausgangsport.

DataAdder Die Komponente **DataAdder** wird alle 100 ms gestartet und holt sich die an den RPorts anliegenden Werte, addiert diese und schreibt das Ergebnis auf den PPort.

DataReceiver Diese Softwarekomponente dient nur dem Empfangen der Werte vom DataAdder. Sie spielt in der Anwendung keine Rolle und wird auch nicht beim Test berücksichtigt.

Interface Allen Ports in der Applikation Adder wurde ein Interface zugewiesen. Dieses sorgt dafür, dass jeder Port den Datentyp Uint8 erwartet.

6.2.2 Systemtest

Hierfür wird die ARXML und die zugehörige RTE dem ArXmlAnalyzer bekannt gemacht. Da die beiden DataProvider selbst Werte erzeugen, muss für einen Systemtest kein Port stimuliert werden. Hier ist, wie bereits erwähnt, das Verhalten des Systems relevant. Es muss lediglich am PPort der SW-C Adder gelauscht werden, um eine Analyse der Daten zu realisieren. Dazu wird das System nach dem erfolgreichen Laden aller Input-Dateien grafisch dargestellt. Auf den PPort der SW-C Adder wird durch einen Rechtsklick die Option „lauschen“ aktiviert. Durch diese Aktion wird der Port grün markiert und gibt dem Nutzer so ein Feedback. Anschließend wird über das Menü „Tools“ der Knopf „SWC-Test“ ausgewählt. Dadurch öffnet sich das Test-Fenster mit allen Testeinstellungen.

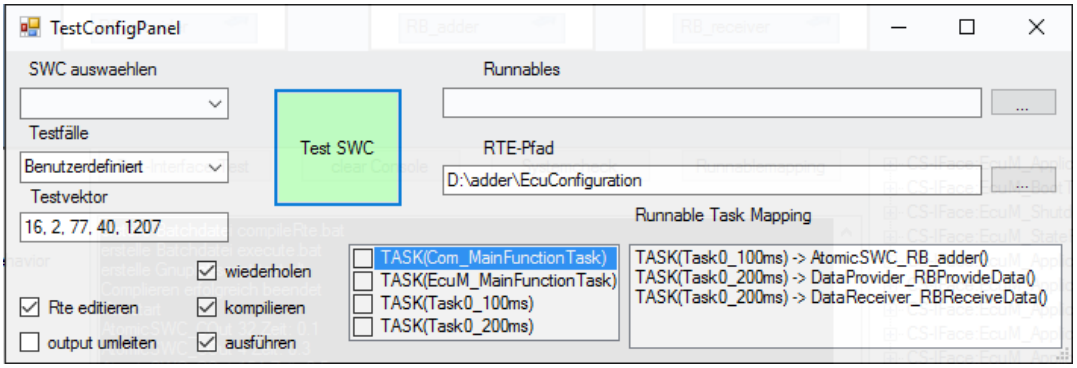

Abbildung 6.6: Testpanel

In der kleinen Listbox (Abbildung 6.6) unten rechts offenbart sich das aus dem statischen Test ermittelte Testresultat. Hier ist die richtige vom RTE-Checker ermittelte Zuordnung zwischen Task und Runnable zu sehen. Die folgende Abbildung zeigt das Ergebnis des Tests.

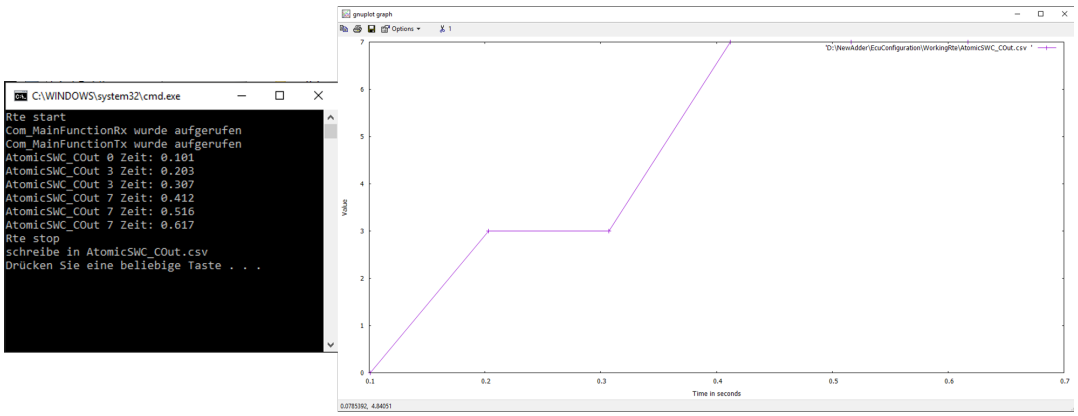

Abbildung 6.7: Testergebnisse Systemtest

Anhand der Ausgabe des Testszenarios kann das Systemverhalten auf Richtigkeit geprüft werden. Weil die Softwarekomponente DataAdder genau wie DataProviderB zum ersten Mal nach hundert Millisekunden gestartet wird, kann der Addierer folglich zu diesem Zeitpunkt auch keine Werte von den beiden DataProvider Komponenten berechnen und verwendet die Initialwerte 0. Das erste Ergebnis an Port **COut** bestätigt dieses Verhalten. Weiterhin ist zu sehen, dass die Zufallswerte des DataProviderA fast immer gleich sind.

6.2.3 Unit-Test einer SW-C

Mit einem Unit-Test soll eine Softwarekomponente einzeln getestet werden. Genaueres ist dem Grundlagenkapitel 2.2.3 Komponententest zu entnehmen. Für einen Komponententest bietet sich die SW-C **DataAdder** an, denn hier soll an den Eingangsports selbst definierte Werte übergeben werden und am Ausgangsport das Resultat überprüft werden. Die Testwerte für den Adder betragen 1, 2, 3, 4, 5, 6. Die Resultate

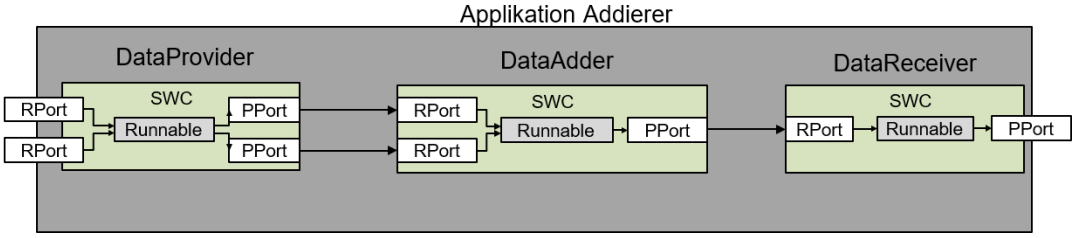

Abbildung 6.8: Unittest von DataAdder

müssen folglich 2, 4, 6, 8, 10, 12 sein. Im Feld Testfälle wird ein benutzerdefinierter

Datenvektor ausgewählt, welcher die oben genannten Testwerte beinhaltet. Durch den Klick auf Test SW-C wird alles generiert was für die Simulation benötigt wird. Anschließend wird automatisiert der Test gestartet. Die folgende Tabelle zeigt die Eingangswerte und die erwarteten Resultate am PPort von DataAdder.

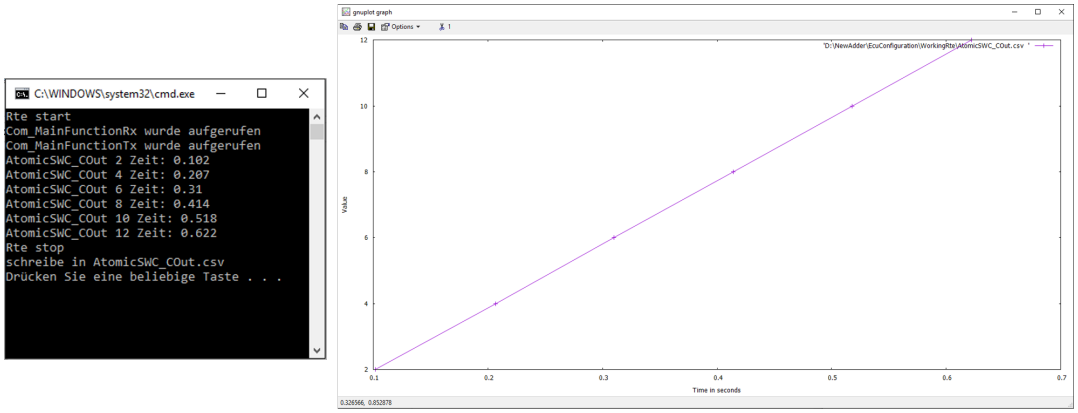

Abbildung 6.9: Testergebnisse Unittest

Die Abbildung zeigt zum einen die durch die Konsole ausgegebenen (links) und zum anderen die von GnuPlot (rechts) dargestellten Ergebnisse. An der Ausgabe ist zu erkennen, dass nach jedem Aufruf eine entsprechende Zeit benötigt wird. Das liegt einerseits an den Berechnungen pro Addition und andererseits am Schreiben der Ergebnisse und Zeitpunktwerte in die entsprechenden Vektoren. Die Tabelle fasst noch

RPortA	RPortB	Soll-Wert PPort	Soll-Zeitwert in ms	Ist-Wert PPort	Ist-Zeitwert in ms
1	1	2	100	2	102
2	2	4	200	4	207
3	3	6	300	6	310
4	4	8	400	8	414
5	5	10	500	10	518
6	6	12	600	12	622

Tabelle 6.2: Resultat und Erwartungswerte

einmal die einzelnen Werte des Tests zusammen. Zu sehen sind die an den PPorts anliegenden Testwerte zum entsprechenden Zeitpunkt. Die Soll-Zeitwerte beziehen sich auf die Ausführungszeiten der Runnable (alle 100 ms), um einen Referenzwert bereitzustellen. Auch hier wird ersichtlich, dass die erwarteten Ergebnisse die Funktionsfähigkeit des Simulators untermauern.

6.3 Zusammenfassung

In diesem Kapitel sollte die Funktionalität dieser Arbeit gezeigt und analysiert werden. Dabei wurde mit den Möglichkeiten der **statischen Analyse** demonstriert, dass Fehler sowohl in der ARXML als auch in den zugehörigen RTE-Dateien aufgedeckt werden können. Um dies zu prüfen, wurde eine mit Fehlern überarbeitete Version der Applikation Adder verwendet, die auch als Grundlage zur Forschung dieser Arbeit diente. Die erzeugten Fehler wurden alle aufgedeckt und an der Konsole ausgegeben.

Das Parsen einer anderen AUTOSAR-Version mit Hilfe des **Profilmanagers** wurde ebenfalls verdeutlicht. Dazu wurde die Applikation Adder in der AUTOSAR-Version 4.4.2 erstellt und die entsprechende ARXML exportiert. Über den Profilmanager wurde eine auf Version 4.4.2 vorbereitete Profildatei importiert und zum Parsen ausgewählt. Auch hierbei stellte es kein Problem dar, alle Elemente der Applikation zu visualisieren. Folglich ist daraus zu schließen, dass alle Informationen eingelesen wurden.

Mit dem **dynamischen Test** konnte die Ausführung des Prüflings in der dafür generierten Simulationsumgebung veranschaulicht werden. Dazu wurde zuerst ein Systemtest realisiert, welcher die richtige Ausführung der Applikation zeigen sollte. Dabei wurde lediglich am Ausgangsport der Softwarekomponenten Adder gelauscht, um so die Richtigkeit des Zeitverhaltens und der Addition selbst zu zeigen. Mit dem Unit-Test einer einzelnen SW-C wurde auch diese Art des Testens beleuchtet. Dabei wurden die Eingangsports durch einen einfachen Mausklick mit Testwerten gespeist und am entsprechenden PPort gelauscht. Auch hierbei waren die erwartenden Ergebnisse richtig.

Die Validierung der Simulation erfolgte hierbei mit einer einfachen Applikation, die zwei Werte zu festgelegten Zeitpunkten miteinander addiert. Den vorher festgelegten Werten wurden vor der Simulation das erwartete Ergebnis zugewiesen und nach der Ausführung mit den tatsächlichen Ergebnissen verglichen. Im Test konnte die korrekte Ausführung der Runnables und das Übergeben und Auslesen der Testdaten nachvollzogen werden. Ein überaus Interessanter Punkt war die Tatsache, dass je größer die Eingangswerte waren, die Ausführung dementsprechend mehr Zeit beanspruchte.

7 Zusammenfassung und Ausblick

Dieses Kapitel fasst noch einmal alle Abschnitte dieser Arbeit zusammen und gibt einen möglichen Ausblick auf Erweiterungen und Verbesserungen.

7.1 Zusammenfassung

Diese Arbeit zeigt ein mögliches Verfahren, einen Simulationsprozess für AUTOSAR-Applikationen mit korrespondierender RTE zu realisieren. Angefangen mit den Grundlagen von ATUTOSAR und den einzelnen Bestandteilen einer Applikation, bis hin zum Verständnis für das Thema Testen und den damit verbundenen Begrifflichkeiten wurden im Grundlagenkapitel alle nötigen Voraussetzungen geschaffen, die im Konzept vorgeschlagenen Lösungen zu verstehen.
Das Kapitel Stand der Technik hat einige der derzeitigen Verfahren beim Testen von AUTOSAR-Applikationen aufgezeigt. Dazu wurden einzelne auf dem Markt befindliche Werkzeuge genauer betrachtet und anhand von vorgegebenen Kriterien auf die Einbezugnahme ins Projekt analysiert.
Im Konzept wurde durch einzelne Phasen jeder Schritt zu einer funktionsfähigen Simulationsumgebung genau beschrieben. Dabei wurde ein konzeptionelles Verfahren vorgeschlagen, welches das Validieren des Inputs und der Verarbeitung der dadurch gewonnen Informationen und die daraus resultierende Simulationsumgebung beschreibt.
Mit dem Abschnitt Implementierung wurden diese konzeptionellen Lösungen entsprechend umgesetzt. Mit Hilfe eines eigens implementierten Profilmanagers und Xml-Parser wurde eine Möglichkeit geschaffen versionsunabhängig ARXML-Dateien einzulesen. Aufgrund der Verwaltung aller Informationen in Form von Objekten, war es möglich alle benötigten Code-Dateien zu erzeugen, um so eine Funktionsfähige Test-Umgebung zu generieren.
Zum Schluss wurde mit dem Kapitel Testergebnisse die Funktionsfähigkeit für eine Simulationsumgebung anhand eines Prototypen demonstriert, in dem diesen Testwerte übergeben und die Ergebnisse dem Nutzer präsentiert werden. Dabei werden sowohl die Ergebnisse als auch die Zugehörigen Zeitwerte ausgegeben. Abgerundet wird das ganze durch die Generierung eines Zeit-Wert Diagramms mit GnuPlot.

7.2 Ausblick

Im Konzept wurden im Kapitel 4.8 einige **Einschränkungen** zu dieser Arbeit aufgezeigt. Ein Simulator soll in der Lage sein, alle AUTOSAR-spezifisch generierten RTE-Dateien zu testen. Dazu muss ein generischer Weg gefunden werden, alle Projektdateien zusammen zu führen, um diese automatisiert kompilieren zu können. Diese liegen je nach Hersteller in verschiedenen Verzeichnissen und könnten z. B. über die Pfadangaben in den Makefiles identifiziert werden. Außerdem sollte es das Ziel sein, das Testobjekt in keinster Weise zu verändern. Dazu könnten mit virtuellen Komponenten die Schnittstellen der zu testenden Softwarekomponenten aufgerufen und die Daten übergeben werden. Andersrum kann durch Bereitstellung der erforderlichen Funktion die Daten abgelauscht werden. Die meisten Möglichkeiten für Erweiterungen bietet die fertig generierte Testumgebung. Mit der im Konzept erwähnten Datei GlobalData.c können alle Testdaten (z. B. Testvektoren) modifiziert oder andere Testdaten hinzugefügt werden. Das Taskverwaltungsmodul kann durch ein verbessertes Scheduler-Modul ergänzt werden. Anhand der Informationen aus der ARXML-Datei kann eine für die betreffende Applikation geeignete Datenstruktur mit Tasks generiert werden. Hierbei können dann Mechanismen zur Vermeidung von konkurrierenden Ressourcen-Zugriffen in kritischen Bereichen (critical section), wie z. B. Speervariablen implementiert werden. Dies wird alles ermöglicht, weil der Kompilier-Prozess in Batch-Dateien abläuft und so vom Nutzer verändert werden kann. D. h. es ist möglich, nachträglich mit zu kompilierende Dateien einzubinden, weil nach dem Erstellen der Simulation eine ausführbare Datei vorliegt, liegt es nahe, diese auf einer ECU auszuführen. Die Applikation könnte dann über den CAN-Bus mit Testwerten versorgt werden. Dazu muss die Simulationsumgebung in der Lage sein, direkt mit der Hardware zu kommunizieren, was normalerweise die Basissoftware ermöglicht.
Die Wichtigsten Erweiterungen betreffen die implementierten Klassenbibliotheken **ArXmlParser** und **RTE-Checker**. Da diese Master-Arbeit nur ein Teil des **ASTAS**-Projekt darstellt, können die genannten Module für weitere Teil-Projekte verwendet und erweitert werden. Der Parser kann durch weitere AUTOSAR-Informationen ergänzt und der RTE-Checker mit weiteren Parse-Algorithmen bestückt werden. Diese würden folglich weiteren Arbeiten zum Thema AUTOSAR zur Verfügung stehen. Zum Schluss ist noch der Profilmanager zu nennen, welcher mit der Möglichkeit eines selbst erstellbaren Profils erweitert werden könnte. Ein spezieller Algorithmus könnte die veränderten Tag-Namen identifizieren und diese in einer separaten Profildatei ablegen.

Literaturverzeichnis

[AUT11] AUTOSAR: *Specification of BSW Scheduler*, 2011.
URL http://www.autosar.org/fileadmin/files/releases/3-2/software-architecture/system-services/standard/AUTOSAR_SWS_BSW_Scheduler.pdf

[aut14a] Hanser automotive: *AUTOSAR erweitert seine Aktivitäten um Acceptance Tests*, *http://www.hanser-automotive.de/aktuelle-messtechnik-news/article/autosar-erweitert-seine-aktivitaeten-um-acceptance-tests.html*, 2014.

[AUT14b] AUTOSAR: *Specification of Communication*, Febr. 2014.
URL http://www.autosar.org/fileadmin/files/releases/3-2/software-architecture/communication-stack/standard/AUTOSAR_SWS_COM.pdf

[AUT14c] AUTOSAR: *Specification of ECU Communication Manager R3.2 Rev 3*, Febr. 2014.
URL http://www.autosar.org/fileadmin/files/releases/3-2/software-architecture/communication-stack/standard/AUTOSAR_SWS_ComManager.pdf

[AUT14d] AUTOSAR: *Specification of ECU State Manager R3.2 Rev 3*, Febr. 2014.
URL http://www.autosar.org/fileadmin/files/releases/3-2/software-architecture/system-services/standard/AUTOSAR_SWS_ECU_StateManager.pdf

[AUT14e] AUTOSAR: *Specification of Operating System*, 2014.
URL http://www.autosar.org/fileadmin/files/releases/3-2/software-architecture/system-services/standard/AUTOSAR_SWS_OS.pdf

[AUT14f] AUTOSAR: *Specification of PDU Router*, Febr. 2014.
URL http://www.autosar.org/fileadmin/files/releases/3-2/software-architecture/

communication-stack/standard/AUTOSAR_SWS_PDU_Router.pdf

[AUT14g] AUTOSAR: *Specification of RTE*, Febr. 2014. URL http://www.autosar.org/fileadmin/files/releases/4-2/software-architecture/rte/standard/AUTOSAR4.2_SWS_RTE.pdf

[AUT14h] AUTOSAR: *Specification of RTE R3.2 Rev 3*, Febr. 2014. URL http://www.autosar.org/fileadmin/files/releases/3-2/software-architecture/rte/standard/AUTOSAR_SWS_RTE.pdf

[AUT14i] AUTOSAR: *Virtual Functional Bus*, März 2014. URL http://www.autosar.org/fileadmin/files/releases/3-2/software-architecture/rte/standard/AUTOSAR_SWS_RTE.pdf

[AUTnd] AUTOSAR: *Overview of Acceptance Tests*, n.d. URL http://www.autosar.org/fileadmin/files/releases/tc-1-1/AUTOSAR_EXP_AcceptanceTestsOverview.pdf

[Cle10] Torsten Cleff: *Basiswissen Testen von Software - Vorbereitung zum Certified Tester (Foundation Level) nach ISTQB-Standard*, W3l GmbH, Witten, 1. aufl. Aufl., 2010, ISBN 978-3-868-34012-9.

[ETA15] ETAS: *ETAS ISOLAR-EVE*, *http://www.etas.com/de/products/isolar$_e$ve−details.php*, 2015.

[Gri13] Rainer Grimm: *C++11 für Programmierer - Den neuen Standard effektiv einsetzen*, O'Reilly Germany, KÃ¶ln, 2013, ISBN 978-3-955-61392-1.

[Har02] Wolfram Hardt: *Integration von Verzögerungszeit-Invarianz in den Entwurf eingebetteter Systeme*, Shaker-Verlag, Achen, 1. aufl. Aufl., 2002, ISBN 3-8265-8251-9.

[HLHA13] Tae Man Han, Jeong-Hwan Lee, Hyun Yong Hwang und Yong Hak Ahn: *A Study on Signal Group Processing of AUTOSAR COM Module*, Techn. Ber., Sejong University, 2013.

[KF09] Olaf Kindel und Mario Friedrich: *Softwareentwicklung mit AUTOSAR - Grundlagen, Engineering, Management in der Praxis*, Dpunkt-Verlag, KÃ¶ln, 1. aufl. Aufl., 2009, ISBN 978-3-898-64563-8.

[Küh13] Andreas Kühnel: *Visual C# 2012 - Das umfassende Handbuch - Spracheinführung, Objektorientierung, Programmiertechniken*, Rheinwerk Verlag GmbH, Bonn, 6. aufl. Aufl., 2013, ISBN 978-3-836-21997-6.

[KT13] Sung Kisoon und Han Taeman: *Development Process for AUTOSAR-based Embedded System*, Techn. Ber., Electronics and Telecommunications Research Institute, 2013.

[KYM+09] Florian Kluge, Chenglong Yu, Jorg Mische, Sascha Uhrig und Theo Ungerer: *Implementing AUTOSAR Scheduling and Resource Management on an Embedded SMT Processor*, Techn. Ber., Department of Compute Science University of Augsburg, 86159 Augsburg, Germany, 2009.

[Rol07] Dieter Roller: *Programmierung in C / C++ - Mit einer grundlegenden Einführung in die Objektorientierung*, expert verlag, Renningen, 1. aufl. Aufl., 2007, ISBN 978-3-816-92629-0.

[Sch14] Ferdinand Schäfer: *Steuergeräte-Entwicklung mit AUTOSAR: Evaluierung der Entwicklungsumgebung Arctic Studio* -, disserta Verlag, Hamburg, 1. aufl. Aufl., 2014, ISBN 978-3-954-25468-2.

[Shp13] Ilya Shpigor: *Instant MinGW Starter* -, Packt Publishing Ltd, Birmingham, 2013, ISBN 978-1-849-69563-3.

[SL05] Andreas Spillner und Tilo Linz: *Basiswissen Softwaretest - Aus- und Weiterbildung zum Certified Tester - Foundation Level nachISTQB-Standard*, Dpunkt-Verlag, KÃ¶ln, 3. überarb. und aktualis. aufl. Aufl., 2005, ISBN 978-3-898-64358-0.

[Von09] Helmut Vonhoegen: *Einstieg in XML* -, Galileo Press GmbH, Bonn, 5. aufl. Aufl., 2009, ISBN 978-3-836-21367-7.

[WEMS+12] Mario Winter, Mohsen Ekssir-Monfared, Harry M. Sneed, Richard Seidl und Lars Borner: *Der Integrationstest - Von Entwurf und Architektur zur Komponenten- und Systemintegration*, Carl Hanser Verlag GmbH Co KG, M, 2012, ISBN 978-3-446-42951-2.

[ZS14] Werner Zimmermann und Ralf Schmidgall: *Bussysteme in der Fahrzeugtechnik - Protokolle, Standards und Softwarearchitektur*, Springer-Verlag, Berlin Heidelberg New York, 5. aufl. Aufl., 2014, ISBN 978-3-658-02419-2.

Selbstständigkeitserklärung

Hiermit erkläre ich, daß ich die vorliegende Arbeit selbstständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendete Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Chemnitz, den 30. März 2016

_____

Markus Leib